O'REILLY®

Migrating Java to the Cloud

Modernize Enterprise Systems Without Starting From Scratch

Kevin Webber &
Jason Goodwin

Migrating Java to the Cloud

Modernize Enterprise Systems without Starting from Scratch

Kevin Webber and Jason Goodwin

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Migrating Java to the Cloud
by Kevin Webber and Jason Goodwin

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Foster
Production Editor: Colleen Cole
Copyeditor: Charles Roumeliotis
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kevin Webber

September 2017: First Edition

Revision History for the First Edition
2017-08-28: First Release

978-1-491-99492-4

[LSI]

Table of Contents

Preface

This book aims to provide practitioners and managers a comprehensive overview of both the advantages of cloud computing and the steps involved to achieve success in an enterprise cloud initiative.

We will cover the following fundamental aspects of an enterprise-scale cloud computing initiative:

- The requirements of applications and infrastructure for cloud computing in an enterprise context
- Step-by-step instructions on how to refresh applications for deployment to a cloud infrastructure
- An overview of common enterprise cloud infrastructure topologies
- The organizational processes that must change in order to support modern development practices such as continuous delivery
- The security considerations of distributed systems in order to reduce exposure to new attack vectors introduced through microservices architecture on cloud infrastructure

The book has been developed for three types of software professionals:

- Java developers who are looking for a broad and hands-on introduction to cloud computing fundamentals in order to support their enterprise's cloud strategy

- Architects who need to understand the broad-scale changes to enterprise systems during the migration of heritage applications from on-premise infrastructure to cloud infrastructure
- Managers and executives who are looking for an introduction to enterprise cloud computing that can be read in one sitting, without glossing over the important details that will make or break a successful enterprise cloud initiative

For developers and architects, this book will also serve as a handy reference while pointing to the deeper learnings required to be successful in building cloud native services and the infrastructure to support them.

The authors are hands-on practitioners who have delivered real-world enterprise cloud systems at scale. With that in mind, this book will also explore changes to enterprise-wide processes and organizational thinking in order to achieve success. An enterprise cloud strategy is not only a purely technical endeavor. Executing a successful cloud migration also requires a refresh of entrenched practices and processes to support a more rapid pace of innovation.

We hope you enjoy reading this book as much as we enjoyed writing it!

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

A deep thanks to Larry Simon for his tremendous editing efforts; writing about multiple topics of such broad scope in a concise format is no easy task, and this book wouldn't have been possible without his tireless help. A big thanks to Oliver White for supporting us in our idea of presenting these topics in a format that can be read in a single sitting. We would also like to thank Hugh McKee, Peter Guagenti, and Edward Hsu for helping us keep our content both correct and enjoyable. Finally, our gratitude to Brian Foster and Jeff Bleiel from O'Reilly for their encouragement and support through the entire writing process.

CHAPTER 1

An Introduction to Cloud Systems

Somewhere around 2002, Jeff Bezos famously issued a mandate that described how software at Amazon had to be written. The tenets were as follows:

- All teams will henceforth expose their data and functionality through service interfaces.
- Teams must communicate with each other through these interfaces.
- There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no backdoors whatsoever. The only communication allowed is via service interface calls over the network.
- It doesn't matter what technology they use.
- All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
- Anyone who doesn't do this will be fired.

The above mandate was the precursor to Amazon Web Services (AWS), the original public cloud offering, and the foundation of everything we cover in this book. To understand the directives above and the rationale behind them is to understand the motivation for an enterprise-wide cloud migration. Jeff Bezos understood

the importance of refactoring Amazon's monolith for the cloud, even at a time when "the cloud" did not yet exist! Amazon's radical success since, in part, has been due to their decision to lease their infrastructure to others and create an extensible company. Other forward-thinking companies such as Netflix run most of their business in Amazon's cloud; Netflix even regularly speaks at AWS's re:Invent conference about their journey to AWS. The Netflix situation is even more intriguing as Netflix competes with the Amazon Video offering! But, the cloud does not care; the cloud is neutral. There is so much value in cloud infrastructure like AWS that Netflix determined it optimal for a competitor to host their systems rather than incur the cost to build their own infrastructure.

Shared databases, shared tables, direct linking: these are typical early attempts at carving up a monolith. Many systems begin the modernization story by breaking apart at a service level only to remain coupled at the data level. The problem with these approaches is that the resulting high degree of coupling means that any changes in the underlying data model will need to be rolled out to multiple services, effectively meaning that you probably spent a fortune to transform a monolithic system into a distributed monolithic system. To phrase this another way, in a distributed system, a change to one component should not require a change to another component. Even if two services are physically separate, they are still coupled if a change to one requires a change in another. At that point they should be merged to reflect the truth.

The tenets in Bezos' mandate hint that we should think of two services as autonomous collections of behavior and state that are completely independent of each other, even with respect to the technologies they're implemented in. Each service would be required to have its own storage mechanisms, independent from and unknown to other services. No shared databases, no shared tables, no direct linking. Organizing services in this manner requires a shift in thinking along with using a set of specific, now well proven techniques. If many services are writing to the same table in a database it may indicate that the table should be its own service. By placing a small service called a *shim* in front of the shared resource, we effectively expose the resource as a service that can be accessed through a public API. We stop thinking about accessing data from databases and start thinking about providing data through services.

Effectively, the core of a modernization project requires architects and developers to focus less on the mechanism of storage, in this case a database, and more on the API. We can abstract away our databases by considering them as services, and by doing so we move in the right direction, thinking about everything in our organization as extensible services rather than implementation details. This is not only a profound technical change, but a cultural one as well. Databases are the antithesis of services and often the epitome of complexity. They often force developers to dig deep into the internals to determine the implicit APIs buried within, but for effective collaboration we need clarity and transparency. Nothing is more clear and transparent than an explicit service API.

According to the 451 Global Digital Infrastructure Alliance, a majority of enterprises surveyed are in two phases of cloud adoption: *Initial Implementation* (31%) or *Broad Implementation* (29%).[1] A services-first approach to development plays a critical role in *application modernization*, which is one of three pillars of a successful cloud adoption initiative. The other two pillars are *infrastructure refresh* and *security modernization*.

Cloud Adoption

Cloud adoption requires careful consideration at all levels of an enterprise. These are the three pillars of a successful cloud adoption:

Infrastructure refresh
: Most organizations will adopt a hybrid-cloud topology, with servers both in the cloud and on the premises acting as a single cluster, and some organizations may also adopt a multi-cloud strategy for flexible deployments across multiple platforms such as AWS, Azure, and GCE, using both containers and VMs.

Application modernization and migration
: Each legacy application must be evaluated and modernized on a case-by-case basis to ensure it is ready to be deployed to a newly refreshed cloud infrastructure.

1 451 Global Digital Infrastructure Report, April 2017.

Security modernization
: The security profile of components at the infrastructure and application layers will change dramatically; security must be a key focus of all cloud adoption efforts.

This book will cover all three pillars, with an emphasis on application modernization and migration. Legacy applications often depend directly on server resources, such as access to a local filesystem, while also requiring manual steps for day-to-day operations, such as accessing individual servers to check log files—a very frustrating experience if you have dozens of servers to check! Some basic refactorings are required for legacy applications to work properly on cloud infrastructure, but minimal refactorings only scratch the surface of what is necessary to make the most of cloud infrastructure.

This book will demonstrate how to treat the cloud as an *unlimited pool of resources* that brings both *scale* and *resilience* to your systems. While the cloud is an enabler for these properties, it doesn't provide them out of the box; for that we must evolve our applications from legacy to *cloud native*.

We also need to think carefully about security. Traditional applications are secure around the edges, what David Strauss refers to as *Death Star security*, but once infiltrated these systems are completely vulnerable to attacks from within. As we begin to break apart our monoliths we expose more of an attack footprint to the outside world, which makes the system as a whole more vulnerable. Security must no longer come as an afterthought.

We will cover proven steps and techniques that will enable us to take full advantage of the power and flexibility of cloud infrastructure. But before we dive into specific techniques, let's first discuss the properties and characteristics of cloud native systems.

What Is Cloud Native?

The Cloud Native Computing Foundation is a Linux Foundation project that aims to provide stewardship and foster the evolution of the cloud ecosystem. Some of the most influential and impactful cloud-native technologies such as Kubernetes, Prometheus, and fluentd are hosted by the CNFC.

The CNFC defines cloud native systems as having three properties:

Container packaged
: Running applications and processes in software containers as an isolated unit of application deployment, and as a mechanism to achieve high levels of resource isolation.

Dynamically managed
: Actively scheduled and actively managed by a central orchestrating process.

Micro-services oriented
: Loosely coupled with dependencies explicitly described (e.g., through service endpoints).

Container Packaged

Resource isolation is the key to building maintainable, robust applications. We can bundle our applications using technologies such as Docker, which allows us to create isolated units of deployment, while also eliminating inconsistencies when moving from environment to environment. With a single command we can build a container image that contains everything that our application requires, from the exact Linux distribution, to all of the command line tools needed at runtime. This gives us an isolated unit of deployment that we can start up on our local machine in the exact same way as in the cloud.

Dynamically Managed

Once we begin to bundle and deploy our applications using containers, we need to manage those containers at runtime across a variety of cloud-provisioned hardware. The difference between container technologies such as Docker and virtualization technologies such as VMWare is the fact that containers abstract away machines completely. Instead, our system is composed of a number of containers that need access to system resources, such as CPU and memory. We don't explicitly deploy *container X* to *server Y*. Rather, we delegate this responsibility to a manager, allowing it to decide where each container should be deployed and executed based on the resources the containers require and the state of our infrastructure. Technologies such as DC/OS from Mesosphere provide the ability to schedule and manage our containers, treating all of the individual resources we provision in the cloud as a single machine.

Microservices Oriented

The difference between a big ball of mud and a maintainable system are well-defined boundaries and interfaces between conceptual components. We often talk about the size of a component, but what's really important is the complexity. Measuring lines of code is the worst way to quantify the complexity of a piece of software. How many lines of code are complex? 10,000? 42?

Instead of worrying about lines of code, we must aim to reduce the conceptual complexity of our systems by isolating unique components from each other. Isolation helps to enhance the understanding of components by reducing the amount of domain knowledge that a single person (or team) requires in order to be effective within that domain. In essence, a well-designed component should be complex enough that it adds business value, but simple enough to be completely understood by the team which builds and maintains it.

Microservices are an architectural style of designing and developing components of container-packaged, dynamically managed systems. A service team may build and maintain an individual component of the system, while the architecture team understands and maintains the behaviour of the system as a whole.

Cloud Infrastructure

Whether public, private, or hybrid, the cloud transforms infrastructure from physical servers into near-infinite pools of resources that are allocated to do work.

There are three distinct approaches to cloud infrastructure:

- A *hypervisor* can be installed on a machine, and discrete virtual machines can be created and used allowing a server to contain many "virtual machines"
- A *container management platform* can be used to manage infrastructure and automate the deployment and scaling of container packaged applications
- A *serverless* approach foregoes building and running code in an environment and instead provides a platform for the deployment and execution of *functions* that integrate with public cloud resources (e.g., database, filesystem, etc.)

VMs on Hypervisors

Installing a hypervisor such as VMWare's ESXi was the traditional approach to creating a cloud. Virtual machines are installed on top of the hypervisor, with each virtual machine (VM) allocated a portion of the computer's CPU and RAM. Applications are then installed inside an operating system on the virtual machine. This approach allows for better utilization of hardware compared to installing applications directly on the operating system as the resources are shared amongst many virtual machines.

Traditional public cloud offerings such as Amazon EC2 and Google Compute Engine (GCE) offer virtual machines in this manner. On-premise hardware can also be used, or a blend of the two approaches can be adopted (hybrid-cloud).

Container Management

A more modern approach to cloud computing is becoming popular with the introduction of tools in the Docker ecosystem. Container management tools enable the use of lightweight VM-like containers that are installed directly on the operating system. This approach has the benefit of being more efficient than running VMs on a hypervisor, as only a single operating system is run on a machine instead of a full operating system with all of its overhead running within each VM. This allows most of the benefits of using full VMs, but with better utilization of hardware. It also frees us from some of the configuration management and potential licensing costs of running many extra operating systems.

Public container-based cloud offerings are also available such as Amazon EC2 Container Service (ECS) and Google Container Engine (GKE).

The difference between VMs and containers is outlined in Figure 1-1.

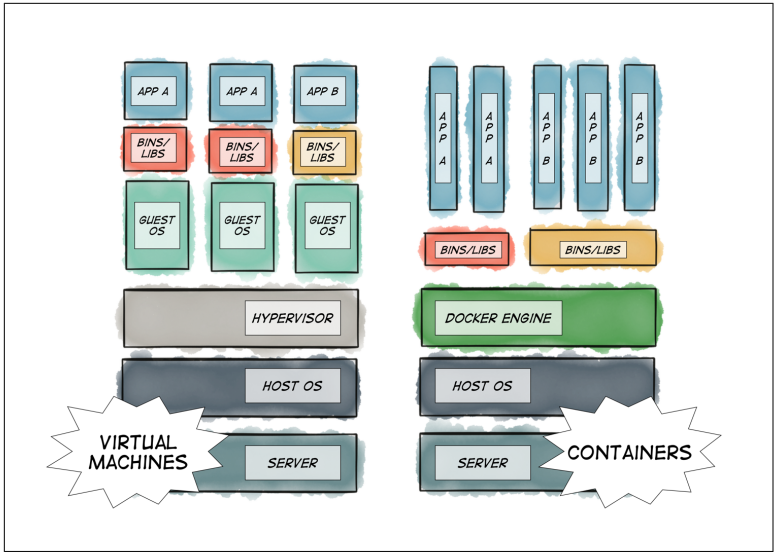

Figure 1-1. VMs, pictured left—many guest operating systems may be hosted on top of hypervisors. Containers, pictured right—apps can share bins/libs, while Docker eliminates the need for guest operating systems.

Another benefit of using a container management tool instead of a hypervisor is that the infrastructure is abstracted away from the developer. Management of virtual machine configuration is greatly simplified by using containers as all resources are configured uniformly in the "cluster." In this scenario, configuration management tools like Ansible can be used to add servers to the container cluster, while configuration management tools like Chef or Puppet handle configuring the servers.

Configuration Management

Container management tools are not used for managing the *configuration* details of servers, such as installing specific command-line tools and applications on each server. For this we would use a configuration management tool such as Chef, Puppet, or Ansible.

Once an organization adopts cloud infrastructure, there's a natural gravitation towards empowering teams to manage their own applications and services. The operations team becomes a manager and

provider of resources in the cloud, while the development team controls the flow and health of applications and services deployed to those resources. There's no more powerful motivator for creating resilient systems than when a development team is fully responsible for what they build and deploy.

These approaches promise to turn your infrastructure into a self-service commodity that DevOps personnel can use and manage themselves. For example, DC/OS—"Datacenter Operating System" from Mesosphere—gives a friendly UI to all of the individual tools required to manage your infrastructure as if it were a single machine, so that DevOps personnel can log in, deploy, test, and scale applications without worrying about installing and configuring an underlying OS.

Mesosphere DC/OS

DC/OS is a collection of open source tools that act together to manage datacenter resources as an extensible pool. It comes with tools to manage the lifecycle of container deployments and data services, to aid in service discovery, load balancing, and networking. It also comes with a UI to allow teams to easily configure and deploy their applications.

DC/OS is centered around Apache Mesos, which is the distributed system kernel that abstracts away the resources of servers. Mesos effectively transforms a collection of servers into a pool of resources —CPU and RAM.

Mesos on its own can be difficult to configure and use effectively. DC/OS eases this by providing all necessary installation tools, along with supporting software such as Marathon for managing tasks, and a friendly UI to ease the management and installation of software on the Mesos cluster. Mesos also offers abstractions that allow *stateful* data service deployments. While stateless services can run in an empty "sandbox" every time they are run, stateful data services such as databases require some type of durable storage that persists through runs.

While we cover DC/OS in this guide primarily as a container management tool, DC/OS is quite broad in its capabilities.

DC/OS Requirements

DC/OS requires a pool of servers, referred to as a *cluster*. There are two types of Mesos servers: *agents* and *masters*. Agents are servers that act as the pool of resources (CPU, RAM) in which to run your applications. There can be any number of agents, and they can be added and removed to expand and shrink the cluster as needed. To manage the agents, there are a few *masters*. Masters use Zookeeper to coordinate amongst themselves in case one experiences failure. A tool called Marathon is included in DC/OS that performs the scheduling and management of your tasks into the agents.

Container management platforms *manage* how resources are allocated to each application instance, as well as how many copies of an application or service are running simultaneously. Similar to how resources are allocated to a virtual machine, a fraction of a server's CPU and RAM are allocated to a running container. An application is easily "scaled out" with the click of a button, causing Marathon to deploy more containers for that application onto agents.

Additional agents can also be added to the cluster to extend the pool of resources available for containers to use. By default, containers can be deployed to any agent, and generally we shouldn't need to worry about which server the instances are run on. Constraints can be placed on where applications are allowed to run to allow for policies such as security to be built into the cluster, or performance reasons such as two services needing to run on the same physical host to meet latency requirements.

Kubernetes

Much like Marathon, Kubernetes—often abbreviated as *k8s*—automates the scheduling and deployment of containerized applications into pools of compute resources. Kubernetes has different concepts and terms than those that DC/OS uses, but the end result is very similar when considering container orchestration capabilities.

DC/OS is a more general-purpose tool than Kubernetes, suitable for running traditional services such as data services and legacy applications as well as container packaged services. Kubernetes might be considered an alternative to DC/OS's container management sched-

uling capabilities alone—directly comparable to Marathon and Mesos rather than the entirety of DC/OS.

In Kubernetes, a *pod* is a group of containers described in a definition. The definition described is the "desired state," which specifies what the running environment should look like. Similar to Marathon, Kubernetes Cluster Management Services will attempt to schedule containers into a pool of *workers* in the cluster. Workers are roughly equivalent to Mesos agents.

A *kubelet* process monitors for failure and notifies Cluster Management Services whenever a deviation from the desired state is detected. This enables the cluster to recover and return to a healthy condition.

DC/OS or Kubernetes?

For the purposes of this book, we will favor DC/OS's approach. We believe that DC/OS is a better choice in a wider range of enterprise situations. Mesosphere offers commercial support, which is critical for enterprise projects, while also remaining portable across cloud vendors.

Going Hybrid

A common topology for enterprise cloud infrastructure is a hybrid-cloud model. In this model, some resources are deployed to a public cloud—such as AWS, GCP, or Azure—and some resources are deployed to a "private cloud" in the enterprise data center. This hybrid cloud can expand and shrink based on the demand of the underlying applications and other resources that are deployed to it. VMs can be provisioned from one or more of the public cloud platforms and added as an elastic extension pool to a company's own VMs.

Both on-premise servers and provisioned servers in the cloud can be managed uniformly with DC/OS. Servers can be dynamically managed in the container cluster, which makes it easier to migrate from private infrastructure out into the public cloud; simply extend the pool of resources and slowly turn the dial from one to the other.

Hybrid clouds are usually sized so that most of the normal load can be handled by the enterprise's own data center. The data center can

continue to be built in a classical style and managed under traditional processes such as ITIL. The public cloud can be leveraged exclusively during *grey sky* situations, such as:

- Pressure on the data center during a transient spike of traffic
- A partial outage due to server failure in the on-premise data center
- Rolling upgrades or other predictable causes of server downtime
- Unpredictable ebbs and flows of demand in development or test environments

The hybrid-cloud model ensures a near-endless pool of global infrastructure resources available to expand into, while making better use of the infrastructure investments already made. A hybrid-cloud infrastructure is best described as elastic; servers can be added to the pool and removed as easily. Hybrid-cloud initiatives typically go hand-in-hand with *multi-cloud* initiatives, managed with tools from companies such as RightScale to provide cohesive management of infrastructure across many cloud providers.

Serverless

Serverless technology enables developers to deploy purely stateless functions to cloud infrastructure, which works by pushing all state into the data tier. Serverless offerings from cloud providers include tools such as AWS Lambda and Google Cloud Functions.

This may be a reasonable architectural decision for smaller systems or organizations exclusively operating on a single cloud provider such as AWS or GCP, but for enterprise systems it's often impossible to justify the lack of portability across cloud vendors. There are no open standards in the world of serverless computing, so you will be locked into whichever platform you build on. This is a major trade-off compared to using an application framework on general cloud infrastructure, which preserves the option of switching cloud providers with little friction.

CHAPTER 2
Cloud Native Requirements

Applications that run on cloud infrastructure need to handle a variety of runtime scenarios that occur less frequently in classical infrastructure, such as transient node or network failure, split-brain state inconsistencies, and the need to gracefully quiesce and shut down nodes as demand drops off.

NOTE

Applications or Services?

We use the term "application" to refer to a legacy or heritage application, and "service" to refer to a modernized service. A system may be composed of both applications and services.

Any application or service deployed to cloud infrastructure must possess a few critical traits:

Location independence
: The ability to move across hosts in the event of a server failure, with no loss of state.

Fast startup
: Cloud nodes are expected to fail and restart, therefore the startup time of services must be ultra fast, ideally measured in single-digit seconds.

Fault-tolerance
: In-memory state will be lost when a node crashes, therefore stateful applications and services that run on cloud infrastructure must have a robust recovery mechanisms.

Reliable communications
: Other processes will continue to communicate with a service or application that has crashed, therefore they must have a mechanism for reliable communications even with a downed node.

Distributed state
: Cloud-native services must support in-memory state and have a mechanism for recovery in case the service fails and restarts.

Infrastructure Requirements

Some so-called cloud native frameworks are completely stateless and depend on pushing all state out of the application layer into either the browser or a database. Without state to maintain in the application itself, it's trivial to provision new resources on demand during a spike of traffic and destroy those resources when traffic subsides. This gives the illusion of being cloud native by a sleight-of-hand rebranding.

Selecting a Cloud Native Framework

The term "cloud native" is so new that vendors are tweaking it to retrofit their existing products, so careful attention to detail is required before selecting frameworks for building cloud native services.

While pushing complexity to another tier of the system, such as the database tier, may sound appealing, this approach is full of risks. Many architects are falling into the trap of selecting a database to host application state in the cloud without fully understanding its characteristics, specifically around consistency guarantees against corruption. Jepsen is an organization that "has analyzed over a dozen databases, coordination services, and queues—and we've found replica divergence, data loss, stale reads, lock conflicts, and much more."

The cloud introduces a number of failure scenarios that architects may not be familiar with, such as node crashes, network partitions,

and clock drift. Pushing the burden to a database doesn't remove the need to understand common edge cases in distributed computing. We continue to require a reasonable approach to managing state—some state should remain in memory, and some state should be persisted to a data store. Let business requirements dictate technical decisions rather than the characteristics or limitations of any given framework.

Our recommendation is to keep as much state as possible in the application tier. After all, the real value of any computer system is its state! We should place the emphasis on state beyond all else—after all, without state programming is pretty easy, but the systems we build wouldn't be very useful.

Automation Requirements

To be scalable, infrastructure must be instantly provisionable, able to be created and destroyed with a single click. The bad old days of physically SSHing into servers and running scripts is over.

Terraform from Hashicorp is an infrastructure automation tool that treats infrastructure as code. In order to create reproducible infrastructure at the click of a button, we codify all of the instructions necessary to set up our infrastructure. Once our infrastructure is codified, provisioning it can be completely automated. Not only can it be automated, but it can follow the same development procedures as the rest of our code, including source control, code reviews, and pull requests.

Terraform is sometimes used to provision VMs and build them from scratch before *every* redeploy of system components in order to prevent configuration drift in the environment's configuration. Configuration drift is an insidious problem in which small changes on each server grows over time, and there's no reasonable way of determining what state each server is in and how each server got into that state. Destroying and rebuilding your infrastructure routinely is the only way to prevent server configuration from drifting away from a baseline configuration.

Even Amazon is not immune from configuration drift. In 2017 a massive outage hit S3, caused by a typo in a script used to restart their servers. Unfortunately, more servers were relaunched than intended and Amazon had not "completely restarted the index subsystem or the placement subsystem in our larger regions for many

years." Eventually the startup issues brought the entire system down. It's important to rebuild infrastructure from scratch routinely to prevent configuration drift issues such as these. We need to exercise our infrastructure to keep it healthy.

> *It is a good idea to virtually burn down your servers at regular intervals. A server should be like a phoenix, regularly rising from the ashes.*[1]
>
> —Martin Fowler

Amazon S3's index and placement subsystem servers were *snowflake servers*. Snowflakes are unique and one of a kind, the complete opposite of the properties we want in a server. According to Martin, the antidote to snowflake servers is to "hold the entire operating configuration of the server in some form of automated recipe." A configuration management tool such as Chef, Puppet, or Ansible can be leveraged to keep provisioned infrastructure configured correctly, while the infrastructure itself can be provisioned and destroyed on demand with Terraform. This ensures that drift is avoided by wiping the slate clean with each deployment.

An end-to-end automation solution needs to ensure that all aspects of the operational environment are properly configured, including routing, load balancing, health checks, system management, monitoring, and recovery. We also need to implement log aggregation to be able to view key events across all logs across all servers in a single view.

Infrastructure automation is of huge benefit even if you aren't using a public cloud service, but is *essential* if you do.

Managing Components at Runtime

Containers are only one type of component that sits atop our cloud infrastructure. As we discussed, Mesosphere DC/OS is a systems management tool that handles the nitty gritty of deploying and scheduling all of the components in your system to run on the provisioned resources.

By moving towards a solution such as DC/OS along with containers, we can enforce process isolation, orchestrate resource utilization, and diagnose and recover from failure. DC/OS is called the "data-

1 Martin Fowler, "PhoenixServer", 10 July 2012.

center operating system" for a reason—it brings a singular way to manage all of the resources we need to run all system components on cloud infrastructure. Not only does DC/OS manage your application containers, but it can manage most anything, including the availability of big data resources. This brings the possibility of having a completely unified view of your systems in the cloud.

We will discuss resource management in more depth in Chapter 4, *Getting Cloud-Native Deployments Right*.

Framework Requirements

Applications deployed to cloud infrastructure must start within seconds, not minutes, which means that not all frameworks are appropriate for cloud deployments. For instance, if we attempt to deploy J2EE applications running on IBM WebSphere to cloud infrastructure, the solution would not meet two earlier requirements we covered: *fast startups* and *graceful shutdowns*. Both are required for rapid scaling, configuration changes, redeploys for continuous deployment, and quickly moving off of problematic hosts. In fact, Zeroturnaround surveys show that the average deploy time of a servlet container such as WebSphere is approximately *2.5 minutes*.

Frameworks such as Play from Lightbend and Spring Boot from Pivotal are stateless API frameworks that have many desirable properties for building cloud-native services. Stateless frameworks require that all state be stored client side, in a database, in a separate cache, or using a distributed in-memory toolkit. Play and Spring Boot can be thought of as an evolution of traditional CRUD-style frameworks that evolved to provide first-class support for RESTful APIs. These frameworks are easy to learn, easy to develop with, and easy to scale at runtime. Another key feature of this modern class of stateless web-based API frameworks is that they support fast startup and graceful shutdowns, which becomes critical when applications begin to rebalance across a shrinking or expanding cloud infrastructure footprint.

Building stateful cloud-native services also requires a completely different category of tool that embraces distribution at its core. Akka from Lightbend is one such tool—a distributed in-memory toolkit. Akka is a toolkit for building stateful applications on the JVM, and one of the only tools in this category that gives Java developers the ability to leverage their existing Java skills. Similar tools include

Elixir, which is programmed in Erlang and runs on the Erlang VM, but they require Java developers to learn a new syntax and a new type of virtual machine.

Akka is such a flexible toolkit for distribution and communications that HTTP in Play is implemented with Akka under the hood. Akka is not only easy-to-use, but a legitimate alternative to complex messaging technologies such as Netty, which was the original tool of choice in this category for Java developers.

Actors for cloud computing

Akka is based on the notion of actors. Actors in Akka are like lightweight threads, consuming only about 300 bytes each. This gives us the ability to spin up thousands of actors (or millions with the passivation techniques discussed in "Leveraging Advanced Akka for Cloud Infrastructure" on page 47) and spread them across cloud infrastructure to do work in parallel. Many Java developers and architects are already familiar with threads and Java's threading model, but actors may be a less familiar model of concurrency for most Java developers. Actors are worth learning as they're a simple way to manage both concurrency and communications. Akka actors can manage communication across physical boundaries in our system—VMs and servers—with relative ease compared to classical distributed object technologies such as CORBA. The actor model is the ideal paradigm for cloud computing because the actor system provides many of the properties we require for cloud-native services, and is also easy to understand. Rather than reaching into the guts of memory, which happens when multiple threads in Java attempt to update the same object instance at once, Akka provides boundaries around memory by enforcing that only message passing can influence the state of an actor.

Actors provide three desirable components for building stateful cloud native services as shown in Figure 2-1:

- A mailbox for receiving messages
- A container for business logic to process received messages
- Isolated state that can be updated only by the actor itself

Actors work with *references* to other actors. They only communicate by passing messages to each other—or even passing messages to themselves! Such controlled access to state is what makes actors so

ideal for cloud computing. Actors never hold references to the *internals* of other actors, which prevents them from directly manipulating the state of other actors. The only way for one actor to influence the state of another actor is to send it a message.

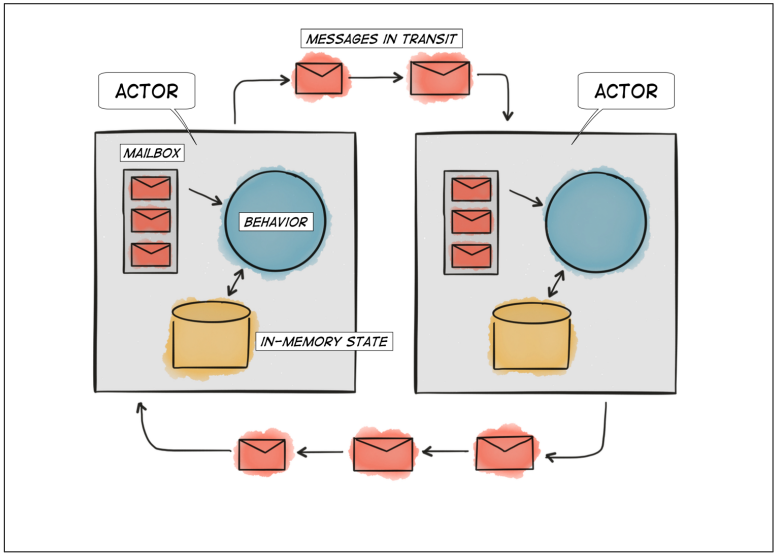

Figure 2-1. The anatomy of an actor in Akka: a mailbox, behavior, and state. Pictured are two actors passing messages to each other.

The actor model was "motivated by the prospect of highly parallel computing machines consisting of dozens, hundreds, or even thousands of independent microprocessors, each with its own local memory and communications processor, communicating via a high-performance communications network."[2] Actors provide developers with two building blocks that are not present in traditional thread-based frameworks: the ability to distribute computation across hosts to achieve parallelism, and the ability to distribute data across hosts for resilience. For this reason, we should strongly consider the use of actors when we need to build stateful services rather than simply pushing all state to a database and hoping for the best. We will cover actors in more depth in "Isolating State with Akka" on page 38.

2 William Clinger (June 1981). "Foundations of Actor Semantics." Mathematics Doctoral Dissertation. MIT.

Configuration

Another critical requirement of our application frameworks is support for immutable configuration. Immutable configuration ensures parity between development and production environments by keeping application configuration separate from the application itself. A deployable application should be thought of as not only the code, but that plus its configuration. They should always be deployed as a unit.

Visibility

Frameworks must provide application-level visibility in the form of tracing and monitoring. Monitoring is well understood, providing critical metrics into the aggregate performance of your systems and pointing out issues and potential optimizations. Tracing is more akin to debugging—think live debugging of code, or tracing through network routes to follow a particular request through a system. Both are important, but tracing becomes much more important than it historically has been when your services are spread across a cloud-based network.

Telemetry data is important in distributed systems. It can become difficult over time to understand all of the complexities of how data flows through all of our services; we need to be able to pinpoint how all of the various components of our systems interact with each other. A cloud-native approach to tracing will help us understand how all components of our system are behaving, including method calls within a service boundary, and messaging across services.

The Lightbend Enterprise Suite includes OpsClarity for deep visibility into the way cloud applications are behaving, providing high-quality telemetry and metrics for data flows and exceptions (especially for Akka-based systems). AppDynamics is another popular tool in this space that provides performance telemetry for high-availability and load-balancing systems.

It's best to configure your applications to emit telemetry back to a monitoring backend, which can then integrate directly with your existing monitoring solution.

Architecture Requirements

In a distributed system we want as much traffic handled towards the edge of the system as possible. For instance, if a CDN is available to serve simple requests like transmitting a static image, we don't want our application server tied up doing it. We want to let each request flow through our system from layer to layer, with the outermost layers ideally handling the bulk of traffic, serving as many requests as possible before reaching the next layer.

Starting at the outermost layer, a basic distributed system typically has a load balancer in front, such as Amazon's Elastic Load Balancer (ELB). Load balancers are used to distribute and balance requests between replicas of services or internal gateways (Figure 2-2).

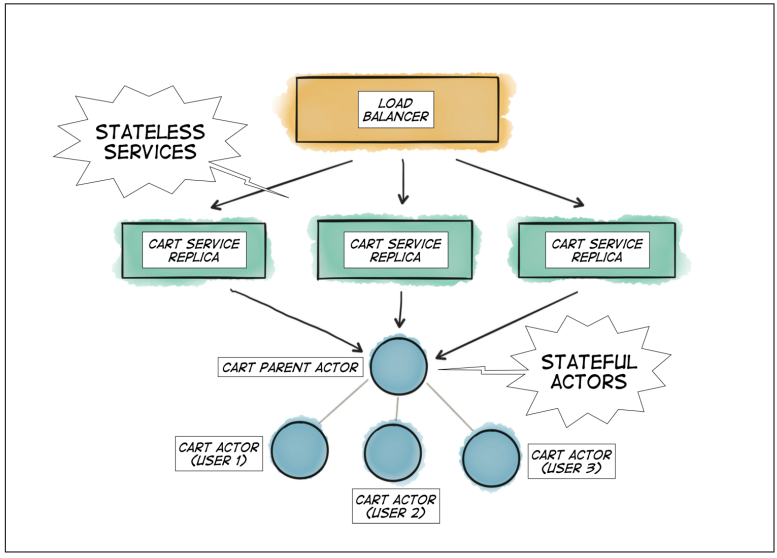

Figure 2-2. A load balancer spreads out traffic among stateless components such as API gateways and services, each of which can be replicated to handle additional load. We have a unique actor for each user's shopping cart, each of which shares the same unique parent actor.

At runtime we can create many instances of our API gateways and stateless services. The number of instances of each service running can be adjusted on-the-fly at runtime as traffic on the systems increases and decreases. This helps us to balance traffic across all available nodes within our cluster. For instance, in an ecommerce system we may have five runtime instances of our API gateway,

three instances of our search service, and only one instance of our cart service. Within the shopping cart's API there will be operations that are stateless, such as a query to determine the total number of active shopping carts for all users, and operations which affect the state of a single unique entity, such as adding a product to a user's shopping cart.

Services or Microservices?

A service may be backed by many microservices. For instance, a shopping cart service may have an endpoint to query the number of active carts for all users, and another endpoint may add a product to a specific user's shopping cart. Each of these service endpoints may be backed by different microservices. For more insight into these patterns we recommend reading *Reactive Microservices Architecture* by Jonas Bonér (O'Reilly).

In a properly designed microservices architecture each service will be individually scalable. This allows us to leverage tools like DC/OS to their full potential, unlocking the ability to perform actions such as increasing the number of running instances of any of the services at runtime with a single click of a button. This makes it easy to scale out, scale in, and handle failure gracefully. If a stateless service crashes, a new one can be restarted in its place and begin to handle requests immediately.

Adding state to a service increases complexity. There's always the possibility of a server crashing or being decommissioned on-the-fly causing us to lose the state of an entity completely. The optimal solution is to distribute state across service instances and physical nodes, which reduces the chance of losing state, but introduces the possibility of inconsistent state. We will cover how to safely distribute state in Chapter 3.

If state is held server side on multiple instances of the same service without distribution, not only do we have to worry about losing state, but we also have to worry about routing each request to the specific server that holds the relevant state. In legacy systems, *sticky sessions* are used to route traffic to the server containing the correct state.

Consider a five-node WebSphere cluster with thousands of concurrent users. The load balancer must determine which user's session is

located on which server and always route requests from that particular user to that particular server. If a server is lost to hardware failure, all of the sessions on that server are lost. This may mean losing anything from shopping cart contents to partially completed orders.

Systems with stateful services can remain responsive under partial failure by making the correct compromises. Services can use different backing mechanisms for state: memory, databases, or filesystems. For speed we want memory access, but for durability we want data persisted to file (directly to the filesystem or to a database). Out of the box, VMs don't have durable disk storage, which is surprising to many people who start using VMs in the cloud. Specific durable storage mechanisms such as Amazon's Elastic Block Store (EBS) must be used to bring durability to data stored to disk.

Now that we have a high-level overview of the technical requirements for cloud-native systems, we will cover how to implement the type of system that we want: systems that fully leverage elastic infrastructure in the cloud, backed by stateless services for the graceful handling of bursts of traffic through flexible replication factors at a service level, and shored up by stateful services so the application state is held in the application itself.

CHAPTER 3

Modernizing Heritage Applications

Monolithic systems are easier to build and reason about in the initial phases of development. By including every aspect of the entire business domain in a single packaged and deployable unit, teams are able to focus purely on the business domain rather than worrying about distributed systems concerns such as messaging patterns and network failures. Best of breed systems today, from Twitter to Netflix to Amazon, started off as monolithic systems. This gave their teams time to fully understand the business domain and how it all fit together.

Over time, monolithic systems become a tangled, complex mess that no single person can fully understand. A small change to one component may cause a catastrophic error in another due to the use of shared libraries, shared databases, improper packaging, or a host of other reasons. This can make the application difficult to separate into services because the risk of any change is so high.

Our first order of business is to slowly *compartmentalize* the system by factoring out different components. By defining clear conceptual boundaries within a monolithic system, we can slowly turn those conceptual boundaries—such as package-level boundaries in the same deployable unit—into physical boundaries. We accomplish this by extracting code from the monolith and moving the equivalent functionality into services.

Let's explore how to define our service boundaries and APIs, while also sharpening the distinction between services and microservices. To do this, we need to step back from the implementation

details for a moment and discuss the techniques that will guide us towards an elegant design. These techniques are called *Event Storming* and *Domain-Driven Design*.

Event Storming and Domain-Driven Design

Refactoring a legacy system is difficult, but luckily there are proven approaches to help get us started. The following techniques are complementary, a series of exercises that when executed in sequence can help us move through the first steps of understanding our existing systems and refactoring them into cloud-native services.

1. *Event Storming* is a type of workshop that can be run with all stakeholders of our application. This will help us understand our business processes without relying on pouring over legacy code—code that may not even reflect the truth of the business! The output of an Event Storming exercise is a solid understanding of business events, processes, and data flows within our organization.
2. *Domain-Driven Design* is a framework we'll use to help us understand the *natural boundaries* within our business processes, systems, and organization. This will help us apply structure to the flow of business activity, helping us to craft clear boundaries at a domain level (such as a line of business), service level (such as a team), and microservice level (the smallest container packaged components of our system).
3. The *anticorruption layer pattern* answers the question of "How do we save as much code from our legacy system as possible?" We do this by implementing *anticorruption layers* that will contain legacy code worth temporarily saving, but ultimately isn't up to the quality standards we expect of our new cloud native services.
4. The *strangler pattern* is an implementation technique that guides us through the ongoing evolution of the system; we can't move from monolith to microservices in one step! The strangler pattern complements the anticorruption layer pattern, enabling us to extract valuable functionality out of the legacy system into the new system, then slowly turning the dial towards the new system allowing it to service more and more of our business.

Event Storming

Event Storming is a set of techniques structured around a workshop, where the focus is to discuss the *flow* of events in your organization. The knowledge gained from an Event Storming session will eventually feed into other modeling techniques in order to provide *structure* to the business flows that emerge. You can build a software system from the models, or simply use the knowledge gained from the conversations in order to better understand and refine the business processes themselves.

The workshop is focused on open collaboration to identify the business processes that need to be delivered by the new system. One of the most challenging aspects of a legacy migration is that no single person fully understands the code well enough to make all of the critical decisions required to port that code to a new platform. Event Storming makes it easier to revisit and redesign business processes by providing a format for a workshop that will guide a deep systems decomposition exercise.

Event Storming by Alberto Brandolini is a pre-release book (at the time of this writing) from the creator of Event Storming himself. This is shaping up to be the seminal text on the techniques described above.

Domain-Driven Design

Domain-Driven Design (DDD) guides us from *flow* to *structure*. DDD introduces us to new terms such as *bounded contexts*, and while the specific terminology may be unfamiliar, the essence of DDD is straightforward. DDD provides us with the essential analysis and design techniques to help us correctly identify the natural boundaries that exist within our systems.

A key goal of our modernization effort is to isolate and compartmentalize components. DDD provides us with all of the techniques required to help us identify the conceptual boundaries that naturally divide components, and model these components as "multiple canonical models" along with their interfaces. The resulting models are easily transformed into working software with very little difference between the models and the code that emerges. This makes DDD the ideal analysis and design methodology for building cloud-native systems.

DDD divides up a large system into Bounded Contexts, each of which can have a unified model—essentially a way of structuring Multiple Canonical Models.

—Martin Fowler

Bounded Contexts in Ecommerce

Products may emerge as a clear boundary within an ecommerce system. Products are added and updated regularly—such as descriptions, inventory, and prices. There are other values of interest, such as the quantity of a specific SKU available at your nearest store. Products would make for a logical bounded context within an ecommerce system, while Shipping and Orders may make two other logical bounded contexts.

Domain-Driven Design Distilled by Vaughn Vernon (Addison-Wesley Professional) is the best concise introduction to DDD currently available.

Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans (Addison-Wesley Professional) is the seminal text on DDD. It's not a trivial read, but for architects looking for a deep dive into distributed systems design and modelling it should be at the top of their reading list.

Refactoring Legacy Applications

According to Michael Feathers, legacy code is "code without tests." Unfortunately, making legacy code serviceable again isn't as simple as adding tests—the code is likely coupled inappropriately, making it very difficult to bring under test with any level of confidence.

First, we need to break apart the legacy code in order to isolate testable units of code. But this introduces a dilemma—code needs to be changed before it can be tested safely, but you can't safely change code that lacks tests. Working with legacy code is fun, isn't it?

Working with Legacy Code

The finer details of working with legacy systems is covered in the book *Working Effectively with Legacy Code* by Michael Feathers (Prentice Hall), which is well worth a read before undertaking an enterprise modernization project.

We need to make the legacy application's functionality explicit through a correct and stable API. The implementation of this *new* API will require invoking the legacy application's *existing* API—if it even has one! If not, we will need to compromise and use another integration pattern such as database integration.

In the first phase of a modernization initiative, the new API will integrate with the legacy application as discussed above. Over time, we will validate our opinions about the true business functionality of the legacy application and can begin to port its functionality to the target system. The new API stays stable, but over time more of the implementation will be backed by the target system.

This pattern is often referred to as the *strangler pattern*, named after the *strangler fig*—a vine that grows upward and around existing trees, slowly "replacing" them with itself.

The API gateway—which we will introduce in detail in the next section—plays a crucial role in the successful implementation of the strangler pattern. The API gateway ensures that service consumers have a stable interface, while the strangler pattern enables the gradual transition of functionality from the legacy application to new cloud native services. Combining the API gateway with the strangler pattern has some noteworthy benefits:

- Service consumers don't need to change as the architecture changes—the API gateway evolves with the functionality of the system rather than being coupled to the implementation details
- Functional risk is mitigated compared to a big-bang rewrite as changes are introduced slowly instead of all at once, and the legacy system remains intact during the entire initiative, continuing to deliver business value
- Project risk is mitigated because the approach is incremental—important functionality can be migrated first, while porting

additional functionality from the legacy application to new services can be delayed if priorities shift or risks are identified

Another complimentary pattern in this space is the *anticorruption layer pattern*. An anticorruption layer is a facade that simplifies access to the functionality of the legacy system by providing an interface, as well as providing a layer for the temporary refactoring of code (Figure 3-1).

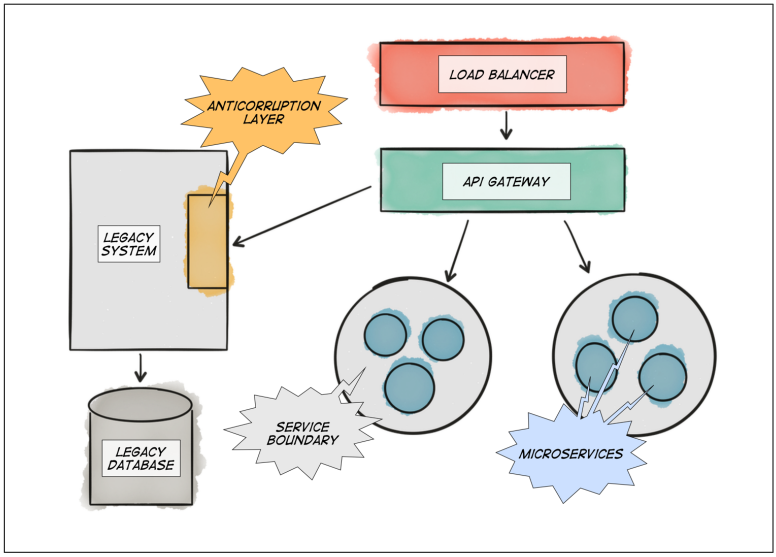

Figure 3-1. A simplified example of an anticorruption layer in a microservices architecture. The anticorruption layer is either embedded within the legacy system or moved into a separate service if the legacy system cannot be modified.

It's tempting to copy legacy code into new services "temporarily," however, much of our legacy code is likely to be in the form of *transaction scripts*. Transaction scripts are procedural spaghetti code not of the quality worth saving, which once ported into the new system will likely remain there indefinitely and *corrupt* the new system.

An anticorruption layer acts both as a *façade*, and also as a transient place for legacy code to live. Some legacy code is valuable now, but will eventually be retired or improved enough to port to the new services. The anticorruption layer pattern is the preferred approach for this problem by Microsoft when recommending how to modernize legacy applications for deployment to Azure.

Regardless of implementation details, the pattern must:

- Provide an interface to existing functionality in the legacy system that the target system requires
- Remove the need to modify legacy code—instead, we copy valuable legacy code into the anticorruption layer for *temporary* use

We will now walk through the implementation of the first layer of our modernized architecture: the API gateway. We will describe the critical role it plays in the success of our new system, and ultimately describe how to implement your own API gateway using the Play framework.

The API Gateway Pattern

An API gateway is a layer that decouples client consumers from service APIs, and also acts as a source of transparency and clarity by publishing API documentation. It serves as a buffer between the outside world and internal services. The services behind an API gateway can change composition without requiring the consumer of the service to change, decoupling system components, which enables much greater flexibility than possible with monolithic systems.

Many commercial off-the-shelf API gateways come with the following (or similar) features:

- Documentation of services
- Load balancing
- Monitoring
- Protocol translation
- Separation of external messaging patterns from internal messaging patterns
- Security (such as ensuring authorization and authentication of callers)
- Abuse protection (such as rate limiting)

Sam Newman, author of *Building Microservices* (O'Reilly), fears that API gateways are becoming the "ESBs of the microservices era." In essence, many API gateways are violating the *smart endpoints and dumb pipes* principle.

Remember, the core purpose of an API gateway is to decouple client consumers from service APIs. Be vigilant against vendor offerings that provide rich logic—that's what services are for! Understanding the role of an API gateway within the wider system architecture will make it easier to make a technical selection decision when evaluating commercial off-the-shelf products.

Smart Endpoints and Dumb Pipes

Many of the features listed above are reasonable for API gateways to offer as they address non-functional requirements. Be especially wary of commercial API gateway products that encourage the implementation of business logic.

API gateways should always be stateless. While the primary goal of an API gateway is to provide a clear interface for our services, the secondary goal of an API gateway is to provide another layer in our architecture to deal with issues of scale and resilience. Stateful components are limited in their ability to scale out and back.

In most cases we do not need to use an off-the-shelf product to implement an API gateway, although some products such as Apigee are available if you prefer something out of the box.

Backends for Frontends

For teams needing to support a number of different user interfaces, such as Android, iOS, and Web, we recommend implementing the *backends for frontends pattern*. This pattern involves creating a unique API gateway per user interface type, enabling UI teams to work independently. For example, this pattern enables the Android team and the iOS team to work independently at their own pace, creating their own gateways with their own unique APIs, mixing and matching backend services as necessary. For teams with a small number of user interface types we recommend using a single top-level API gateway instead of the more complex backends for frontends pattern.

Implementing an API Gateway with Play

Play is a Model-View-Controller (MVC) framework built on top of the JVM (Figure 3-2). Unlike traditional Java web frameworks, Play does not follow the servlet standard. This makes it undesirable and unnecessary to deploy Play to a traditional servlet container like Tomcat, but ideal to deploy to cloud infrastructure, either directly on a JVM or container packaged with Docker.

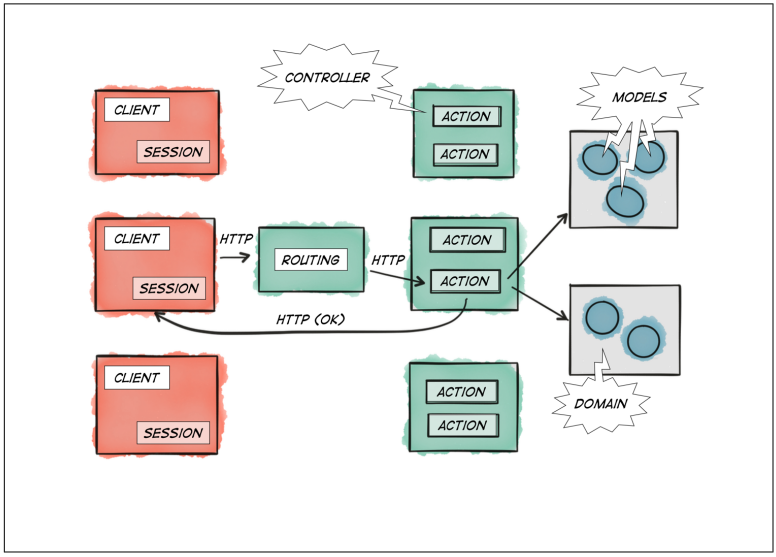

Figure 3-2. The anatomy of Play is similar to most modern MVC frameworks. Play routes a request to the action specified in the routes file and composes a response from the domain. Responses may be rendered in formats such as JSON or HTML.

NOTE

View Rendering

For the purposes of this book we are assuming you are not using Play for view rendering but rather implementing a modern Javascript framework such as React or Angular.

Play is made up of a few key building blocks:

Routing engine
: Dispatches HTTP/REST calls to the appropriate controller action.

Actions
: Functions implemented in the controller tier to process HTTP requests.

Models
: Core business logic (not to be confused with the Active Record Pattern).

Views
: An optional component that helps developers to easily render HTML.

Creating the API

An Event Storming exercise should help us to define a clean API based on the needs of our business. At the end of an Event Storming workshop we should have a list of commands and events that represent our business processes, which we can use to guide us towards the definition of service endpoints in code. DDD will help us to define the boundaries in our API, which become separate controllers in Play.

We will demonstrate a simplified example ecommerce API below.

Routes

Play makes it very easy to expose a simple API through routes (Example 3-1):

Example 3-1. An example Play routes file for a basic ecommerce API.

```
# Products
GET  /api/product/:id  controllers.Product.getProduct(id: String)
GET  /api/products     controllers.Product.getProducts()

# Orders
GET  /api/order/:id    controllers.Order.findOrder(id: String)
PUT  /api/order/:id    controllers.Order.checkout(id: String)

# Cart
GET     /api/cart/:id  controllers.Cart.getCart(id: String)
DELETE  /api/cart/:id  controllers.Cart.deleteCart(id: String)
PUT     /api/cart/:id  controllers.Cart.updateCart(id: String)
```

In the preceding example, we already have a number of bounded contexts identified with their associated interfaces. We can see the product, order, and cart contexts, each with endpoints that map directly to business processes.

Controllers and models

Before we can evolve and change the technology used in our system, we must first isolate the business domain logic. It's very common for applications to begin their lives as a collection of transaction scripts where logic and database access are mixed into the controller. As complexity increases, these types of solutions become more expensive and difficult to work with. Most older enterprise systems have logic spread through the controller and some type of service or application layer.

Our ultimate goal is to isolate the *core domain*—pure business logic—from the infrastructure, controller, and database code. Dependencies should only exist from those layers *inwards* toward the core domain model, as described in an *onion architecture*.

The essence of an onion architecture is that requests are *enriched* with information needed from the outmost layers as they move inwards towards the core domain. For example, you may have an HTTP layer that logs requests, generates correlation identifiers, and so forth. Requests flow inwards towards the core, enriched with the necessary data from the outer layers (Figure 3-3). The primary value of this pattern are the pure cores that emerge, each core containing only business logic.

Implementing the onion architecture successfully makes it easy to refactor a legacy application into independent, isolated services. It also creates a well-defined layer within our system that is tech agnostic, only depending on the programming language that it was implemented in.

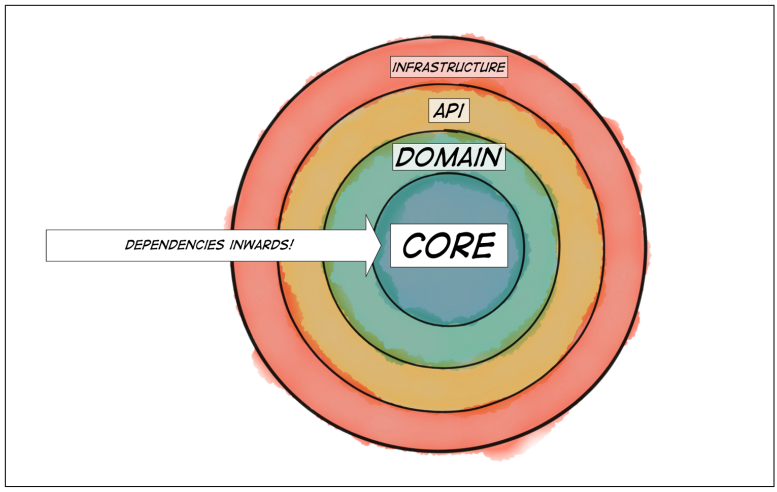

Figure 3-3. The onion architecture helps to reduce coupling within applications. Requests flow inwards—requests are enriched with the data required by the inner layers. Inner layers should never call outwards!

We must map the onion terminology to Play's terminology and package structure:

- The *infrastructure layer* contains supporting code that is framework specific or specific to the infrastructure, such as health checks.
- The *API layer* serves as the public interface to a domain, handling routing and controller logic along with all other requirements of an API gateway.
- The *domain* roughly maps to a bounded context—in ecommerce, "Product" would be the right level for the domain.
- The *core* contains the implementation of our business logic.

Our first order of business is to implement a rigorous package structure for our Play application that follows the guiding principles of onion architecture (Example 3-2).

Example 3-2. Onion architecture expressed in the package structure of a monolithic Play application for ecommerce. Over time, the core of each domain can be factored out into individual microservices.

```
├── controllers
└── domain
|   └── cart
|   |   ├── api
|   |   └── core
|   └── order
|   |   ├── api
|   |   └── core
|   └── product
|       ├── api
|       └── core
└── infrastructure
```

Each domain above will have a corresponding controller, while business logic for each domain will be implemented in the core packages. Assuming this application starts as a monolith, the `cart`, `order`, and `product` packages will initially serve two purposes: exposing an API for the domain, and implementing business logic directly in each core. If the application is refactored into a microservices architecture, business logic is removed from each core, and the core acts as a proxy for the new microservice(s).

Real-World Microservices

Monolithic applications are not all bad. In fact, it's often *better* to begin with a monolith before optimizing towards microservices. It gives us a chance to evolve the API and change implementation details as business processes and domains become better understood by the team, especially in the beginning phases of a new project. Think of microservices as a refactoring technique.

Each domain is made up of two distinct packages, `api` and `core` (Example 3-3), although the core package may have a more descriptive name (such as `cluster`). Following the principles of the onion architecture, only the `api` package should be shared outside of the domain. This keeps the implementation of business logic loosely coupled by preserving the ability to refactor cores into their own microservices.

Example 3-3. An example Product domain in Play.

```
└── domain
    └── product
        ├── api
        │   ├── Price.java
        │   ├── Product.java
        │   └── ProductService.java
        └── core
            └── ProductServiceImpl.java
```

Dependency Injection

Regardless of the framework you use, it's strongly recommended that cores be *dependency injected* rather than hard-wired. Without dependency injection it's difficult to implement an onion architecture.

We've implemented `api` and `core` folders for each domain, but we should not think of Play as the ideal location to hold any state. It's only acceptable to retain state in memory in Play when in development phases, perhaps while still exploring the business domain. But before moving Play into production for any nontrivial system, we must factor out all state into another tier. As we've discussed, Play can handle issues of scale and resilience only as long as it remains stateless—enabling us to instantiate a (virtually) unlimited number of Play instances behind a load balancer, scaling out and in on demand.

We will encounter a situation where we need to work with stateful cores, such as a unique customer with items in their shopping cart. In this situation, we will need to refactor the cores into their own services. Play will then serve as a *proxy* for stateful entities, and the entities will be implemented with Akka actors.

For more advice on building Play applications, we highly recommend reading *Play in Practice* by Will Sargant (O'Reilly).

Isolating State with Akka

Most cloud-native frameworks do not allow a way to retain state other than through a database, or by pushing it out to the client. Akka, on the other hand, is all about state. Not only does Akka provide a mechanism for dealing with state on cloud infrastructure, but

it also provides a level of resilience not possible with most other technologies.

We will demonstrate how to handle state in a cloud-native system without resorting to cookies or the database by leveraging Akka. Often times, state belongs in the application layer itself, and the only legitimate reason to force it to other tiers of our architecture is because the framework provides no other way to deal with it. Akka makes managing state much easier, so we will cover it from first principles.

Do You Need Akka?

If you're building a web-facing system that expects low traffic volumes and can handle a few hours of downtime here and there, you do not need Akka. Akka is a tool for organizations who are building stateful *high availability* (HA) systems that need to scale.

The architecture and programming model of Akka is different than what Java developers are commonly accustomed to. This is for good reason. The very nature of the cloud requires a completely different approach to design and architecture. The reason technologies like CORBA failed is that they tried to mask distribution, whereas Akka makes distribution explicit and manages it appropriately.

When communicating between actors, Akka commonly uses the *tell* pattern to fire off a message to another actor (Example 3-4). An alternate messaging pattern in Akka is *ask*. `tell` sends a message without waiting for a response, while `ask` sends a message and awaits a reply.

Example 3-4. The logic executed when Akka receives an UpdateCart command. This sends a "done" message to the sender of the message.

```
persist(msg, (UpdateCart m) ->
{
   setCartItems(m.getCartItems());
   sender().tell("done", null);
});
```

In the example Akka code above, we accomplish three things whenever we receive an `UpdateCart` command:

1. We persist the message to the backing journal *(persist)*.
2. We update the state of our actor *(setCartItems)*.
3. We signal to the sender that we're done *(tell)*.

The `persist` code block may seem quite unfamiliar. Effectively, we're persisting all `UpdateCart` commands to Cassandra (or another backing journal of your choosing). Once a message is stored in Cassandra—if configured properly—it is *replicated* across a number of hosts on cloud infrastructure. This provides assurances that messages will not be lost even in the event of server failure. (The *replication factor* is how many replicas store a set.) This means that we will never lose persisted messages—even if a server crashes! This also means that the state of the actor is resilient, able to withstand many types of failure, but always able to recover from the journal. This can be an even safer mechanism for the durable storage of state than relational databases, which we will explain in detail.

Actor state can't be changed outside of receiving a message, so we:

Increase the consistency of our system by isolating individual components
: Actors introduce physical boundaries around the state in our system, which prevents the uncontrolled manipulation of memory by outside threads or processes.

Increase the resilience of our system
: We can rebuild our state by replaying the messages that we stored in the distributed journal if we need to move the actor instance to another node (for instance, if the original node fails).

Increase the efficiency of our system
: The code above is asynchronous. `sender()` is a reference to whoever sent us the message. They aren't waiting for a reply, so they're busy doing other work. We simply send them a message back when we're done.

The actor model itself only addresses the basic foundation for inter-process communication through message passing. Actors in Akka bring three other critical properties of a cloud-native system:

- Location transparency
- Failure detection
- Failure handling

Programmers do not need to work directly with the mailbox, rather, when an actor receives a new message it is passed to its *receive* method. Let's look at how a shopping cart actor behaves when it receives a message (Example 3-5).

Example 3-5. A receive method in Akka. Incoming messages are matched to their corresponding class definitions. Specific business logic is executed for each message type.

```
@Override
public Receive createReceive() {
  return receiveBuilder()
    .match(EmptyCart.class, msg ->
        persist(msg, (EmptyCart m) ->
        {
            emptyCart();
            sender().tell("done", null);
        }))
    .match(UpdateCart.class, msg -> {
        persist(msg, (UpdateCart m) ->
        {
            setCartItems(m.getCartItems());
            sender().tell("done", null);
        });
    })
    .match(GetContents.class, msg -> {
        sender().tell(cartItems, self());
    })
    .matchEquals(ReceiveTimeout.getInstance(), msg -> passivate())
    .matchAny(msg -> System.out.println("unknown message: " + msg))
    .build();
}
```

In Example 3-5, we can see that for each type with a matching handler, we execute the logic specifically for that message type. `UpdateCart` has its own logic, as does `GetContents`. The message instances themselves are serialized and easily passed over the wire. Once the message is matched to a handler, it is easy for a developer

to do the other things that actors do best: manipulate state, send messages, or create new actors.

Akka Persistence

You may have noticed the `persist` code blocks in Example 3-5. This is an implementation of *Akka Persistence* and *event sourcing*, both of which we will cover in detail later in this chapter.

Last but not least, how does an actor obtain a reference to another actor? Quite simply: it requests a reference from the runtime (Example 3-6).

Example 3-6. Play requests a reference to the cart actor for a specific user and sends it an UpdateCart message.

```
@Override
public CompletionStage updateCartItems(String userId,
                                       List<CartItem> cartItems) {
  return PatternsCS.ask(shardRegion,
                        new UpdateCart(userId, cartItems), 10000);
}
```

In Example 3-6, we wish to update the shopping cart contents of a particular user. We ask the Akka runtime to do this for us. By providing the `userId` to Akka, it can then locate the shopping cart for that particular user. The user's shopping cart is represented by a unique, stateful actor managed by *Akka Cluster*. The `ask` pattern, implemented above, signifies that we expect a message back. *Ask* is an alternative messaging pattern to *tell*, which does not expect a reply.

One of the most important properties of Akka actors are that they are *location transparent*. That means that an actor can communicate with another actor over the network (or within the same VM) without knowing where the other resides. The decision on whether an actor is local or remote is specified in configuration, but the Akka runtime ultimately decides where the actor lives. The runtime is able to detect when the infrastructure changes—such as when a node goes down—and react accordingly, for example, by killing actors on the unhealthy node and restarting them on a healthy node. This is how Akka enables developers to create resilient cloud-based applications: in the cloud, resilience is everything.

Actors create and pass around *references* to other actors, or even references to themselves. Because actors don't need to worry about the physical location of other actors, the runtime is free to move the physical actors around the infrastructure, but the references to other actors remains stable. This means that one actor can be sending messages to another, even while the other actor is in the middle of crashing on a dying node and being restarted on a healthy node. Akka actors are specifically designed to deal with state, but more importantly, state in a completely asynchronous, highly concurrent, and potentially unstable environment.

Learning Akka

For a graceful introduction to Akka, we recommend *Learning Akka* by Jason Goodwin (Packt Publishing). For a more thorough look into Akka, we encourage you to read the official Akka documentation, which explains the details of *process isolation*, *actor hierarchies*, *supervision*, and every other feature of the toolkit.

The rest of this book will focus on some of the more advanced features of Akka that make it so ideal for building cloud-native systems. We highly recommend visiting the documentation (and a good book) to complement the rest of the material we cover.

Let's explore some of the design patterns behind the code examples we've covered. Akka matches on messages it receives, but what *are* those messages, and how do we go about defining them?

Let's begin with *event-driven architecture* (EDA). EDA guides us towards the correct design for messages with Akka or any other message passing toolkit. Our messages should only come in two forms: *commands* and *events*.

A command is the intention to change the state of an entity, while an event is a the fact of the system. Returning to our running example of a shopping cart, it's intuitive to identify a simple flow of commands and events that the average shopping cart must implement, such as *UpdateCart*, *CartUpdated*, and so forth. Commands and events are so intuitive that once a team learns the heuristics it's not even necessary to include the words "command" and "event" in naming conventions. The secret is *time:* commands are what we

indent to happen in the *future*, and events are facts that happened in the *past*.

How Soon Is Now?

We can look at commands as *intent* and events as *facts*, but we should never think in terms of *now*. In distributed systems there are few guarantees of things happening immediately without fail.

We will explore two EDA-related patterns, giving us even more options for crafting elegant actor-based systems: *event sourcing* and *CQRS*. Then we will dive into additional advanced features of Akka that make implementing these patterns possible in real-world systems.

Event Sourcing

> *Instead of storing just the current state of the data in a domain, use an append-only store to record the full series of actions taken on that data. The store acts as the system of record and can be used to materialize the domain objects.*[1]
>
> —Azure documentation

Rather than simply storing only the most recent state of our shopping cart, we also record the full series of actions—in the form of messages—that have been taken on the actor's state. Based on Microsoft's definition of event sourcing above, in Akka the domain object is the Akka actor itself (Example 3-7).

Example 3-7. An example of event sourcing.

```
persist(evt, (CartUpdated e) ->
{
   setCartItems(msg.getCartItems());
   sender().tell("done", null);
});
```

Observant readers will notice that in previous examples of `persist` we were persisting *commands* rather than *events*. The previous examples were implementations of *command sourcing*, which is a

1 Microsoft Azure"Event Sourcing pattern", 2017-6-23.

subtle variation of event sourcing. You are free to use either flavor of this pattern as long as usage remains consistent.

Event sourcing brings us tremendous benefits not only in the cloud but in general use cases. Not only do we know the *current* state of our shopping cart, but we can also determine *exactly how it got there* by reviewing the log of messages in our append-only store. The event store effectively becomes the "system of record" rather than a database! While this may seem like a profound change for many Java developers, this is how many mainframe systems have worked for decades. For instance, think of a system that handles personal checking accounts; as checks are deposited and funds withdrawn, the system doesn't simply update the client's account balance. It records the transactions in a journal and *derives* the balance from the journal.

Recommended Reading

To learn more about event sourcing, we highly recommend *Implementing Domain-Driven Design* by Vaughn Vernon and *Event Centric* by Greg Young.

Separate Read Models with CQRS

Command query responsibility separation (CQRS) is a common pattern in modern scalable architectures where separate and distinct services are used for *commands*, which affect an entity, and for *queries*, which return the state of that entity, as shown in Figure 3-4. Separating commands and queries allows independent tuning of reads and writes, which often have very different nonfunctional requirements.

At the cost of added complexity, isolating queries from commands by placing them in their own services gives great flexibility. We can change and optimize how data is modeled for reads, because reads (queries) and writes (commands) have independent models. As an example, a product in ecommerce might consist of several different entities such as a product description, a price, and a quantity available. A product might be read many times a second but may only be written to a few times in its lifetime. And we might need to search for products based on the text in the description. In this case we might choose to store a product's true record in a relational database, but use ElasticSearch for queries.

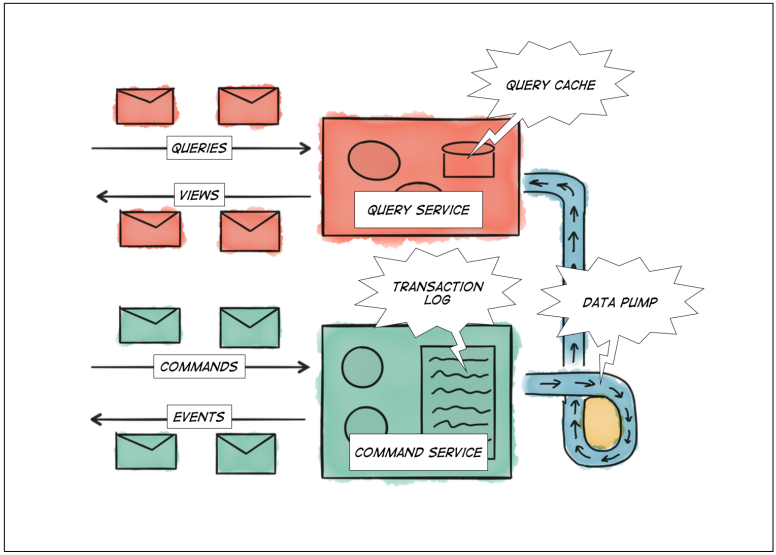

Figure 3-4. For query-intensive services, a separate service called a data pump extracts views from the command-side and caches them on the query side.

CQRS is a natural fit for event sourcing, and is often used in conjunction with it, although the two can be applied separately just as easily. Martin Fowler warns that CQRS can introduce complexity, but we have seen it used successfully in high scale video streaming platforms. If used with approaches like event sourcing, code with CQRS becomes easier to understand as only commands related to business logic will exist in the core of the domain model.

This sums up two important patterns for building event-driven systems that are both scalable and resilient. There is no requirement to use event sourcing and CQRS within your services, but when needed these patterns can provide a tremendous advantage.

Now we are ready to talk about some of Akka's more advanced features. Akka Persistence provides the backbone for implementing event sourcing with Akka, and Akka Cluster enables full location transparency—actors moving around nodes within our cloud infrastructure, without our code having to worry about their physical location.

Leveraging Advanced Akka for Cloud Infrastructure

The combination of Akka Persistence and Akka Cluster, along with the architecture we outlined throughout the modernization section, enables developers and architects to craft stateful, scalable, resilient services that are genuinely cloud native.

Akka Persistence

Akka Persistence is an event sourcing implementation that moves the source of truth from a datastore into the application by maintaining a persistent log of events as they occur. If an application crashes, it is able to recover the state of any lost actors by replaying the past events.

Event sourcing with Akka Persistence offers several benefits over traditional CRUD approaches that modify records in a datastore:

- Easier to build rich domain models, often resulting in better code quality.
- Locking and transactional concerns can be simpler.
- Race conditions in read/write operations are harder to introduce.
- The entire history of state changes can be read and replayed to understand and debug behavior.

The biggest difference between CRUD and event sourcing with Akka Persistence is that once a change is made in a traditional database, it's hard to understand what caused the change, and it's even more difficult to rewind and play back all changes in a consistent manner. Traditional databases are mutable, supporting destructive operations such as update and delete, while append-only logs are a complete record of all mutations. Individual records cannot be updated or deleted. Updates and deletes are handled by appending a record of the change, so updates and deletes become logical rather than physical. This makes bringing the source of truth into the application even safer than pushing out state to a mutable database! The application holds the current state, while the transaction log offers a resilient record of how the state evolved over time.

Remember the `persist` code blocks in our earlier code examples? `persist` signals that we are using Akka Persistence. When we receive a message, we must follow three steps:

1. Check if we can apply the event successfully
2. Persist the event to our backing journal
3. Apply the event to the state of our actor

This allows us to replay events if we experience node failure and must restart the actor on another node.

Recovering from failure is fairly straightforward. All that's required is to extend `AbstractPersistentActor` and implement a `createReceiveRecover` method with logic on how to replay all of the stored messages in the transaction log (Example 3-8).

Example 3-8. An example of recovering actor state with Akka Persistence.

```
public class Cart extends AbstractPersistentActor {
  List<CartItem> cartItems = List.empty();
  // additional actor logic
  @Override
  public Receive createReceiveRecover() {
    return receiveBuilder()
      .match(EmptyCart.class, msg -> emptyCart())
      .match(UpdateCart.class, msg ->
             cartItems = msg.getCartItems())
      .build();
  }
  // additional actor logic
}
```

Akka Persistence also offers mechanisms such as *passivation* that lets you put an actor to sleep when it has no activity in order to save memory. This means that if an actor receives no activity over a certain period of time, it is purposefully removed from memory. If at some point in the future a message is sent to a passivated actor, it is awoken and brought back to its correct state by replaying all messages since the last checkpoint from the transaction log—no different than if the actor had crashed. By passivating actors, we can achieve a nearly limitless amount of scale, handling potentially millions of stateful actors without running out of resources. Passivation can be thought of as a form of scaling up—making the most of the resources we already have available.

Snapshots

When a persistent actor recovers, it is first offered a *snapshot* in order to rebuild its state. Without snapshots, recovery time for an actor can be significant.

Akka Cluster

A cluster is a group of networked computers that's logically treated as a single machine. A distributed system is a collection of services on one or more clusters, which communicates through message passing. It's not uncommon for a modern distributed system to have clusters of hundreds of networked machines to provide a single conceptual service. But before we discuss Akka Cluster in depth, we need to cover what an *actor system* is.

Actors do not exist alone, they exist in a *system*. When an actor receives a message, during processing it can also create child actors, making it responsible for the health of its children. All actors within the system are addressable based on their logical path, which is hierarchal based on ancestry (in essence modeling parent/child relationships) (Figure 3-5).

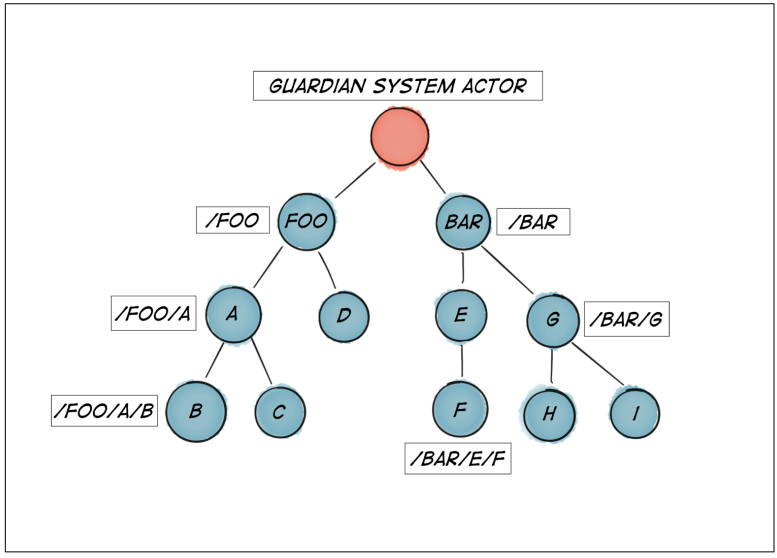

Figure 3-5. Actors come in systems. Parents are responsible for supervising their children and handling failures. Actors are addressable by hierarchy and name.

An *actor system* is like a single application. If an actor system is deployed to cloud infrastructure, it will exist on a single JVM at runtime and all communication between actors will be local. So, what if we want to scale out an actor system by *clustering* multiple actor systems together, allowing the actors to spread out and communicate across VM and network boundaries as if they were still local? That's where Akka Cluster comes in.

Imagine we have cloud infrastructure with thousands of hosts available. Using Akka Cluster, we can treat each one of the actor system instances as a member node, joining them together to form a single cluster—even if the member nodes exist on physically separate machines. Any actor instance that exists within any member node becomes addressable as if it were local. Once clustered, it does not matter whether `/foo/a` exists on the same VM, on a different VM on the same host, on a different host, or a different datacenter than another actor. A sender will always be able to obtain a stable reference to `/foo/a` no matter where it lives—Akka knows where all actors live at all times. When you send a message to the reference of `/foo/a`, Akka looks up the physical location of the actor and routes the message accordingly.

Akka Cluster gives us the ability to scale across any number of servers. As servers are added and removed, Akka Cluster will migrate entities through the cluster in response. With Akka Persistence, this also brings the ability for a member node existing on a server to crash—for example, if the server itself crashes—and be restarted on another host within our infrastructure. Once the actor system rejoins the cluster, all of the individual actors return to their proper state by recovering from the transaction log.

Now that we have discussed how to keep application state within the application tier itself using Akka, we can cover different database and data store options for the cloud and discuss the appropriate use cases for them.

Integration with Datastores

There are many uses of databases, but holding all application state in a database should be a red flag. After all, we enjoyed the use of variables within our web applications *before* the cloud, and we should enjoy the flexibility of continuing to do so *in* the cloud. We simply

need to be careful about how state is managed by respecting how the infrastructure behaves.

Traditional RDBMS: Amazon RDS or Google Cloud SQL

For traditional RDBMS services, AWS and Google offer "databases as a service," freeing operations from managing databases on the premises. They offer cross-datacenter replication and failover to help manage availability, while offering open source engines that work with your existing software with minimal modification. Because these are basic RDBMS protocols, your existing code will continue to function with them.

Unfortunately, even if run asynchronously, JDBC calls will block a thread in your application, which can cause poor performance when there are many users making calls to the database. To isolate JDBC-oriented performance problems, it's recommended that you run them in a separate thread pool. Specifically, we would recommend dispatching JDBC calls to a thread pool dedicated only for database calls, and allocate a moderate number of threads to the pool.

NewSQL

There has been a trend toward new relational databases designed to work at global scale, especially in the wake of the Google Spanner paper, which has prompted efforts such as the open source CockroachDB. Google Cloud Spanner is a public relational database service based on Google's internal Spanner database. Spanner was described internally at Google as "the crown jewel of Google's infrastructure." It promises high availability with transactional consistency at global scale by using a blend of proprietary hardware, such as atomic clocks, and software to produce a synchronously replicated relational database. Consistency and availability (CA in CAP) together are impressive claims for services to make, and most datastores that make such claims fail to deliver around the edge cases. Impressively, Google has used the technology internally and is now making it available to the public, so you can be confident that it has been proven to work.

Similarly, Amazon announced Aurora a few years ago. Aurora is a "cloud-native relational database as a service" which operates in a cluster across availability zones, promising greater availability and performance than more traditional cloud RDBMS services. Aurora

uses a MySQL engine, so it is easy to use with existing applications. In comparison, Spanner now has JDBC drivers but has some non-standard SQL syntax.

NoSQL

NoSQL, as an alternative to relational datastores, is a general term meant to include any denormalized datastore, such as a key-value store or columnar nonrelational datastore. There are a few important datastores that are seeing wide use, and each has a different use case they fit. As microservices are autonomous and have different needs, it's very common to see many different datastore technologies in use, with each service choosing the technologies that fit its use cases.

For ephemeral data and caching, such as session information, Redis is a popular choice. It's a strongly consistent in-memory key-value datastore, similar to services such as Memcached. It has no availability guarantees across node failures but is a handy "Swiss Army knife" with many useful features for coordination between services, such as blocking queues, simple pub-sub, and set-if-not-exist and time-to-live semantics on records which can be used to create locks as a very rudimentary solution for handling contention.

Cassandra is another popular NoSQL datastore. It's a columnar, eventually consistent datastore that acts on "tables" which are *sharded* and replicated across nodes in a cluster. A popular approach for its use is to tune it toward consistency by replicating a record across three nodes, and requiring the acknowledgment of at least two writes (quorum writes) and the reads from two nodes (quorum reads). This lets you lose one of every three nodes in a cluster without interruption in service while making strong consistency guarantees.

Event sourcing uses a datastore as an append-only journal, so Cassandra is an excellent match for event sourcing use-cases.

Consistency and Conflicts in Cassandra

The formula for determining consistency as a promise is:

```
replicationFactor < nodesRead + nodesWritten
```

Note that Cassandra uses wall-clock time with last write wins if there are conflicts during *updates*. This means that it may drop records during conflicting writes and so is safest for use as an immutable store. See the excellent post from Aphyr for further information about this characteristic of Cassandra.

CHAPTER 4
Getting Cloud-Native Deployments Right

In traditional enterprise deployments, it's typical to have only a few machines and a single shared database. We assume the server and database IP will never change. We may use a *virtual IP address* (VIP) to dynamically reroute traffic in the event of a failure, but for the most part keep things simple with hardcoded or config file–based static IPs.

Before a major deployment, everyone sits in a room and discusses the plan before submitting the change request. The Change Management team reviews all of the scheduled changes to ensure the change won't interfere with other activities in the organization. They'll approve the deployment after reviewing all of the artifacts in a *Change Advisory Board* (CAB) meeting.

To redeploy the application, everyone gets on a conference call in the middle of the night and starts the deployment process by changing the *httpd* configuration to display a maintenance page. Two hours later, after manually making changes to the database schema, and modifying some configuration by hand, everyone congratulates each other on a job well done, hangs up, and goes back to bed. As we fall asleep, we pretend everything is fine; there's no way we made a few mistakes when we manually edited the environment, right?

"On the bright side, even if the worst happens it's not like I own the company. I can always find another job."

But the first time an enterprise team redeploys their systems in the middle of the day with no maintenance window is a special event.

"We're deploying right now. We're deploying in the middle of the day!"

Nobody can make a mistake with the configuration at the last minute because all changes—every aspect of deployment from start to finish—is codified, reviewed, tested, and automated. The processes are tested and stable, so we know that they will behave exactly the same way every time. Nobody logs into a server. In many cases, organizations will completely disable *ssh* access to production environments. This is what it means to have operational maturity.

Once a team reaches a high level of operational maturity it's very difficult to introduce human error into the deployment equation. A successful deployment is always a click away, it takes only a few minutes—no downtime, no maintenance windows. The redeployment job removes a node from the load balancer, redeploys, and adds the node back to the load balancer, waiting for application health to be confirmed before moving onto the next node.

In this section we're going to outline the best practices when deploying services, and also discuss integration of these new methodologies into the organization's established processes.

Organizational Challenges

Before we get to the deployment best practices themselves, it's worth highlighting some of the difficulties you'll likely encounter when refreshing development and operational practices in larger, established organizations.

Most large organizations employ a set of "good practices" described in the Information Technology Infrastructure Library (ITIL). Practices such as change management are described in ITIL, and technology leadership across the globe believes that implementation of these practices is a yardstick to measure maturity of the technical organization.

Unfortunately, change management practices can be at odds with many of the modern deployment practices we've seen in the most innovative technology companies. In past years, monthly or even quarterly releases were the norm (and continue to be the norm in many enterprises).

With the explosion in popularity of *Agile software development* practices, development teams started to move toward sprint-based releases, with a single release at the end of each sprint whether that is weekly or bi-weekly. Teams would ensure that whatever code was checked in could be released at any point in time. Radical innovation in practices such as continuous delivery drop the cost of redeploys, and with good development practices and mature deployment pipelines, risk falls as well. According to a survey by New Relic, many organizations are deploying daily or even multiple times per day and the trend is toward more frequent deployments over time.

In change management practices, to *log into a database and manually add a row* is considered a "change." To *interact with an application that adds a row* is not considered a "change" as the application has already been tested and approved for release.

To understand continuous deployment with respect to change management, we can frame our activities such that changes to the deployment pipeline should be treated as a change, but once the deployment pipeline is in place, deployments themselves are a piece of software functionality. A deployment is not a "change" then—it is a piece of functionality in deployed software. Changes to application code are trickier to fit into change management processes, so you may have to meet your organization in the middle.

Political challenges can also be difficult to overcome. There will be many people within an organization who are invested in the status quo. Clayton M. Christensen goes even further in his seminal book *The Innovator's Dilemma* (HarperBusiness), describing that innovation within an organization is often treated like an "invasive species that the organization will attack like an immune system response, even if the attack of innovation causes the failure of the organization."

The unknown is scary to most people. Changing the status quo can be threatening and feel "personal," especially for those who were responsible for defining the status quo. It can be difficult to navigate political waters until you prove that the innovations you evangelize are effective. The role of a leader is not simply technical, it is also evangelism and navigating corporate politics. If you believe in your vision, you must build alliances and work towards small victories, using those initial victories to grow your alliances and further innovation. Only these activities will cause change within a large organi-

zation. This is why organizations move so slow: the most talented technical people within an organization can be the most resistant to politicking. But it's essential to *inspire* change rather than *force* change—the latter is not effective.

Deployment Pipeline

If legacy infrastructure and deployment schedules are simple and infrequent enough, manual deployments may not be a significant cost problem. Even with human error taken into consideration, if deployments are infrequent and exist within a maintenance window it may not be logical to build new automated deployment capabilities. We can SSH into servers, run a few scripts, and call it a day. This is possible because we target infrequent—maybe quarterly—deployments, and the topology of our deployments is static.

When we have hundreds of servers in different geographic areas running dozens of different services with servers that change daily, the scale of the problem grows exponentially. It's no longer reasonable to assume someone will SSH onto each server to deploy code and manually execute scripts, let alone the day-to-day tasks like pulling exception logs from individual servers. Even keeping a server inventory up-to-date is unreasonable at this point. What we need is a reliable mechanism to manage the entire lifecycle of our systems, including infrastructure, code, and deployments. In this scenario, the goal from day one should be to eventually disable SSH access to all servers. In order to do that, you need to consider all reasons why you would log on to a server, and then automate those tasks.

Abstractions exist that help us deal with the complexity of enterprise-scale infrastructure. More traditional VM infrastructure might have "instance groups" that boot machines from "instance templates," while container orchestration platforms such as DC/OS and Kubernetes completely abstract away infrastructure from our applications and services.

Both paths are viable—it's up to you and your organization to decide what works the best. A lot of innovation has gone into container deployments and related clustering platforms, so it would be advisable to consider them even if they are unfamiliar in your organization.

Infrastructure Automation

Before you deploy your applications, you should consider how your infrastructure is managed. Building, scaling, and removing servers should not be a human task.

Whether you're leaning towards hypervisors or container management platforms, your best bet is to manage infrastructure the same way as you manage code. This might seem counterintuitive in a world where operations and developers live in their own worlds, but the emerging DevOps culture has created an "organizational Venn diagram" where the two disciplines overlap and collaborate.

If you're logging on to servers and making manual changes, then you'll certainly introduce "configuration drift" through human error. Even if you maintain servers for a long time, if you stop servers from being updated by hand, it's possible to get very close to reasonably maintaining the same servers for a very long time without drift by using "idempotent updates" of the servers. Tools like Ansible are incredibly easy to pick up and learn for people of varying skill levels, and the resulting playbooks can be reviewed like code. "Pull requests" are made against infrastructure and configuration management code bases, where all stakeholders have a chance to review and catch any issues before they are promoted to live environments.

"Immutable servers" are preferred where servers are regularly decommissioned and rebuilt instead of simply updating the running infrastructure. As discussed, this type of approach will ensure beyond any doubt that the running servers accurately reflect the configuration that has been committed.

Tools like DC/OS and Kubernetes can make these approaches clearer by ensuring that the container is mostly isolated from the environment.

The only variants between two instances of the same contained packaged application are:

- The configuration of the application inside the container *(immutable application configuration)*
- The configuration of the container itself (e.g., *the configuration expressed in a Dockerfile*)

Both variants are clearly expressed in code when working with containers and container management tools like DC/OS or Kubernetes, which means there should no potential of configuration drift.

Expressing infrastructure as code should be a top priority for teams in complex environments, especially as additional resources are provisioned to expand the capacity of the cluster.

Configuration in the Environment

It's tempting to treat configuration as an afterthought. After all, it's not the hottest area of modern computing, but it's an important one. A look at post-mortems shows that configuration is a common cause of incidents.

Best practices related to distributed systems are outlined in *The Twelve-Factor App*. Twelve-Factor Apps pass configuration to the application from the environment. For instance, the IP of the database is not added to the application's configuration, but instead it is added to the environment and referenced from a variable. Success is measurable. For instance, can the same artifact running in a preproduction environment be copied to the production servers? If the answer is no, explore what decisions have been made that limit this portability, and research if environment variables can mitigate the issue.

For example, in DC/OS and Kubernetes, it's quite easy to create and reference environment variables. The Typesafe Config library can be declared to use a variable in the environment or else fall back to development mode variables (Example 4-1).

Example 4-1. Typesafe Config library example of using environment variables along with a fallback mechanism to use development mode configuration if the environment variables are not present.

```
hostname = "127.0.0.1"
hostname = ${?HOST}
```

The second variable is noted as optional with the `?` character. `127.0.0.1` will be used if no `$HOST` variable is present in the environment. This allows a developer to boot the application locally in test mode, while allowing a startup script installed in the environment or container management task scheduling tool such as Mara-

thon to pass in variables. As long as tooling in the deployment chain considers these qualities, it's possible to use environment variables for configuration.

Artifacts from Continuous Integration

The build artifacts need to be stored somewhere for deployment. The decision to use either containers or traditional virtualization infrastructure will change how the artifacts are stored and deployed.

If you're using VMs, you'll likely store the application build artifacts after successful builds in your *continuous integration* (CI) tool such as Jenkins or Bamboo. After building the application, it's common to have an automatic deployment into a test environment. From there, it's easy to take the artifact and "promote" it to further environments —eventually to QA. Assuming the configuration is stored in the environment, you never need to build an artifact specifically for production.

As mentioned, if you're using traditional VM infrastructure, we recommend destroying and recreating servers with a tool such as Terraform instead of trying to manage the servers over multiple releases with a tool such as Ansible alone. Ansible promises idempotent changes, but there is a risk of "drift" that may have some impact eventually.

If you're using Kubernetes or DC/OS, then the process is slightly different:

1. The end of the build process from CI pushes the container to a private Docker registry.
2. The cluster task—for example, Marathon configuration or Pod definition—is modified with the new docker image from the build task.
3. The platform will note that the state of the configuration is different than the running task so it will begin a rolling restart.

Promotion of an artifact is done by updating the Marathon or Pod definition in the next environment.

Logs

Logs should be sent to a central location where they can be searched via a log aggregation mechanism such as the ELK stack. The ELK

stack is powered by ElasticSearch and fed logs by Logstash, an agent that runs on each machine to collect and transport logs from individual servers. Finally, Kibana rounds out ELK by providing a friendly UI for querying, navigating, and displaying log information in ElasticSearch.

Be Aware of Existing Tools

It's common for enterprises to have their own log aggregation tools such as Splunk, so it's important to do full research of what already exists in your organization before advocating for the introduction of new tools.

A topic worth highlighting is the importance of a standardized log format. If extra information, such as correlation (trace) IDs or user IDs, is inserted into log lines across all services in a uniform manner, tools like Logstash can ensure that those entries are correctly indexed. Standardizing the log format across all applications and services allows easier insight into system-wide logs by providing uniform navigability of all log entries.

Autoscaling

Rules can be put in place to automatically add extra resources to our infrastructure when needed. Traffic patterns in production systems are often unpredictable, so the ability to add resources on the fly becomes critical to achieve our goal of implementing scalable, elastic systems.

For example, some of the record-setting Powerball lotteries in the United States have caused unexpectedly large quantities of traffic to lottery websites as everyone rushes to see if they hit the jackpot! To ensure that services stay available—at a reasonable cost—they need to be scaled out and back in as traffic patterns change. Otherwise, teams will need to provision more hardware than required at the highest possible peak of traffic, which at most times will be an enormous waste of money as resources go underutilized.

In the case of VMs, images can be used to bootstrap an application so that provisioning resources doesn't require human intervention. You'll need to ensure that all build processes and autoscaling pro-

cesses trigger the exact same "infrastructure as code" and "configuration as code" mechanisms to reliably configure the servers.

Tools such as DC/OS and Kubernetes can be configured to allocate more resources to a service when CPU utilization or requests per second thresholds are exceeded. You'll need to consider how to expand the cluster itself if it reaches a critical point, by adding more servers to the public cloud (assuming a hybrid-cloud topology).

As has been mentioned, your applications need to start quickly to ensure that autoscaling happens before application health is impacted by rapid spikes in traffic.

Scaling Down

Removing nodes, it turns out, is a lot harder then adding them. We must be tolerant of some failures occurring. We want to get as close to zero failures as possible during scale-down events and redeployments, although realistically we must be tolerant of a few dropped requests.

The native way to deal with shutdown signals is to send the process a `SIGTERM` to notify the application that it should clean up and shutdown. For applications that are running in containers, Docker will issue a `SIGTERM` to your application when the container starts to shut down, then wait a few seconds, and then send a `SIGKILL` if the application has not shut itself down. By building your applications around `SIGTERM` you ensure that they can run anywhere. Some tools use HTTP endpoints to control shutdown, but it's better to be infrastructure agnostic and prefer `SIGTERM` as the signal for graceful shutdown.

Akka Cluster improved its graceful shutdown significantly in Akka 2.5, so developers don't necessarily need to implement specific shutdown logic. Actors move between running nodes in the cluster during shutdown, so there may be a very small window when requests in flight won't hit the referenced actor, but an Akka cluster will migrate and recover from graceful shutdown quickly.

For external-facing applications you want to "drain" nodes before shutting down by removing the application from the load balancer gracefully, and then waiting a short period of time for any requests in-flight to finish. By implementing node draining logic you will be

able to scale down or redeploy in the middle of the day without dropping a single user request!

For DC/OS in particular, if you're using HTTP/REST, we would recommend you look at using the Layer 7 Marathon-LB and trying the Zero Downtime Deployment script (*zdd.py*) for redeploys as the mechanism of choice from your CI tool.

Service Discovery

A key difference in cloud computing compared to more traditional deployments is that the location of services may not be known and cannot be easily managed manually. Heritage enterprise applications typically have a configuration file for each environment where services and databases are described, along with locations such as IP addresses.

Approaches of managing IPs in configuration can be problematic in cloud services. If servers restart in cloud deployments, or if you scale up or down, then IPs can change or otherwise be unknown to already running applications so a service discovery mechanism is needed that will work more reliably. There are a few approaches that can be used depending on the infrastructure strategy selected.

Service Registry

Services can register themselves in a datastore such as Zookeeper or Consul. These key-value stores were built for this type of problem. They can take care of managing a session to clean up any nodes that crash ungracefully, as well as notifying all watchers of any changes in the state of running applications.

As an example, both Kafka and DC/OS Mesos Masters use Zookeeper for discovering and coordinating running nodes. These solutions are reliable and battle-tested, but they do require writing some code in the client for registration and discovery. Client libraries like Curator from Netflix have many common recipes built in to make this relatively simple and to eliminate errors.

Do Not Share Zookeeper Clusters

Although Kafka or DC/OS may come with Zookeeper out of the box, in order to isolate incidents it's recommended that you run a separate Zookeeper cluster for service discovery.

DNS or Load Balancers

DNS or load balancers (LBs) can be used to aid in service discovery. In certain cases you need to know the exact location of a server, so this will not fit all use cases. However, TCP traffic can be routed through a level 4 LB, so this approach is generic enough to work with most applications.

Kubernetes and DC/OS have their own service discovery abstractions that give a name to a service to allow its members to be discovered using a blend of DNS (to find the load balancer) and layer 4 or layer 7 load-balancing mechanisms.

Cluster/Gossip

Some tools such as Cassandra or Akka require only that a "seed node" be discoverable and then the cluster will "gossip" about its other members to keep information about the dynamic state of the cluster up-to-date.

If running on traditional infrastructure or on a hypervisor it can be easier to maintain a few seed nodes for use in discovering the service. In Kubernetes or DC/OS it may be easier to use another service discovery mechanism to find another running node first.

Tools exist to aid in managing the bootstrapping of clusters in different environments. For example, in DC/OS there is a Cassandra package that will use Mesos's DNS to ease discovery of running Cassandra nodes. For Akka Cluster, an open source library called ConstructR exists that can aid in seed node discovery. Lightbend also has a commercial tool, ConductR, that can aid in managing your running cluster on container orchestration platforms such as DC/OS or Kubernetes.

Cloud-Ready Active-Passive

A similar problem to service discovery is leader election. Often a single instance of an application will be responsible for a task—batch processing in an enterprise for example—where having multiple running instances would cause data corruption or other race conditions.

Rather than deploying a single instance, the application can be modified so that one instance becomes the "leader" and the other instances sit on standby to fill in in case the leader fails.

Tools like Zookeeper and Consul can be used to implement *leader election* in order to ensure that only a single node considers itself the leader of a given task. In the case of failure, the other nodes will be notified and will elect a new leader to take over the work. This is a useful pattern in porting existing systems that need to be the only instance doing any processing (*singleton instances*).

Alternatively, Akka Cluster has a mechanism called "Cluster Singleton" that will offload the leadership election concern to Akka. This ensures that only one of the "singleton actors" are running in the cluster at any given time. This has the same effect as implementing your own leadership election logic, but relies on gossip instead of a coordination service.

While this approach can be useful for porting existing logic, if building services from scratch—especially services that are publicly consumed—it's often better to avoid leadership election approaches and instead use other methods that allow work sharing to ensure better availability and scalability.

Failing Fast

Inside an actor system, Akka will respond to unexpected failure by killing and restarting an actor and dropping the message that caused the failure. For many unexpected application-local errors, this is a reasonable approach as we cannot assume that either the message or the state of the actor are recoverable. Similarly, in distributed systems, there are situations in which intentionally crashing the application is the safest approach.

It might seem logical to try to gracefully handle failure inside an application but often taking a pessimistic approach is safer and

more correct. As an example, if a lock is held in a remote service such as Zookeeper, a "disconnect event" for that service would mean that the lock is in an unknown state. The event could be a result of a long GC pause for example—so we can't make any guarantees about how much time has passed or what other applications are doing with the expired lock.

While the chance of a pathological situation may be low, given enough servers running for long enough those pathological scenarios become more likely to be encountered. As we can't make any assumptions about what other applications have done after the lock expiry, the safest response to this type of situation is to throw an exception and shut down the node. If a system is built in a resilient and fault-tolerant manner, then the shutdown of the application will not have a serious impact. It should be acceptable for a few requests to be dropped without causing a catastrophic failure or unrecoverable inconsistencies.

When applications crash, it's important that they are restarted automatically. Kubernetes and DC/OS will handle these events by noting that the process has died, and will attempt to restart it immediately. If deploying into an OS, a wrapper should be used to ensure the process runs. For example, in Linux, `serviced` is often used to start and monitor containers or processes, handling any failures automatically. An application crash should never require manual intervention to restart.

Split Brains and Islands

One of the biggest risks in running stateful clustered services is the risk of "split brain" scenarios. If a portion of nodes in a cluster becomes unavailable for a long period of time, there is no way for the rest of the services running to know if the other nodes are still running or not. Those other nodes may have had a *netsplit*, or they may have had a hard crash—there's no way to know for sure.

It's possible that they will eventually become accessible again. The problem is that, if two sides of a still running cluster are both working in isolation due to a netsplit, they must come to some conclusion of who should continue running and who should shut down. A cluster on one side of the partition must be elected as the surviving portion of the cluster and the other cluster must shut down. The worst-case scenario is that both portions of the cluster think they are

the cluster that should be running, which can lead to major issues, such as two singleton actors (one on each side of the partition) continuing to work, causing duplicated entities and completely corrupt each other's data.

If you're building your own clustered architecture using Akka, Akka's Split Brain Resolver (SBR) will take care of these scenarios for you by ensuring that a strategy is in place, such as *keep majority* or *keep oldest*, depending on whether you wish to keep the largest of a split cluster or the oldest of a split cluster. The strategy itself is configurable. Regardless of what strategy you choose, with SBR it's possible that your entire cluster will die, but your solution should handle this by restarting the cluster so the impact will be minimized.

As much of a risk as *split brain* is the possibility of creating *islands*, where two clusters are started and are unaware of each other. If this occurs with Akka Cluster, it's almost always as a result of misconfiguration. The ConstructR library, or Lightbend's ConductR, can mitigate these issues by having a coordination service ensure that only one cluster can be created.

When using another clustered and stateful technology, such as another framework or datastore, you should carefully evaluate if split brain or island scenarios are possible, and understand how the tool resolves such scenarios.

Split brain and island scenarios can cause data corruption, so it's important to mitigate any such scenarios early and carefully consider approaches for prevention.

Putting It All Together with DC/OS

We're looked at some of the concerns and approaches related to enterprise deployments, and covered how applications behave at runtime. We will conclude with a quick view into what an end-to-end example might look like through to delivery with DC/OS as a target platform.

First, code is stored in a repository. It's common for smaller organizations to use GitHub in the public cloud, but generally enterprise organizations keep code within the safety of the company network, so a local installation of GitLab or GitHub Enterprise might be used instead.

Whenever code is checked in, it will trigger a CI tool such as GitLab's built-in continuous integration functionality—or a separate tool such as Jenkins—to check out and test the latest code automatically. After the CI tool compiles the code a Docker image will be created and stored in a *container registry* such as Docker Hub or Amazon's ECR (Elastic Container Repository).

Assuming that the test and build succeeds, DC/OS's app definition (expressed in a Marathon configuration configuration file) in the test environment is updated by the CI tool pointing Marathon to the location of the newest Docker image. This triggers Marathon to download the image and deploy (or redeploy) it.

The configuration in Example 4-2 contains the Docker image location, port, and network information (including the ports that the container should expose). This example configuration uses a fixed host port, but often you'll be using random ports to allow multiple instances to be deployed to the same host.

Example 4-2. Example Marathon configuration.

```
{
    "id": "my-awesome-app",
    "container": {
    "type": "DOCKER",
    "docker": {
          "image": "location/of/my/image",
          "network": "BRIDGE",
          "portMappings": [
            { "hostPort": 80, "containerPort": 80, "protocol": "tcp"}
          ]
        }
    },
    "instances": 4,
    "cpus": 0.1,
    "mem": 64,
    "upgradeStrategy": {
      "minimumHealthCapacity": 1,
      "maximumOverCapacity": 0.3
    }
}
```

This example is of a basic configuration, but contains everything needed to have Marathon deploy an application into the cluster. The configuration describes the horizontal and vertical scale qualities of the deployments—the number of instances and the resources pro-

vided to each of those instances. You'll note there is no information about where the applications are going to run: there are no provisioning servers, no logging onto servers, no Linux versions, no configuration, and no dependencies. This overlap between ops and development, and the enablement of teams that these solutions provide, highlights the value of using container orchestration frameworks.

Ideally, Marathon should be configured to not drop below the current number of nodes in the deployment. It will start a redeploy by adding a small number of nodes, waiting for them to give a readiness signal via a *health check endpoint*.

Likewise, you don't want Marathon to start too many services all at once. These behaviors are described in the `upgradeStrategy` section in Example 4-2. This works by configuring the `minimumHealthCapacity` and `maximumOverCapacity` values. If `minimumHealthCapacity` is set to `1`, Marathon will ensure the number of servers never falls below 100% of the number of instances configured during redeploys. `maximumOverCapacity` dictates how many nodes should start up at a time. Defining the value at `.3`, we instruct Marathon to not replace more than 30% of the instances at a time. Once the new batch of nodes are up and running and return a 200 status code from the health check endpoints configured, Marathon will then instruct Docker to shut down, which causes `SIGTERM` to be issued to the application, and after a configurable time, `SIGKILL` if the process has not shut down before exceeding the configured threshold. Marathon will continue to replace nodes like this until all are running in the new cluster.

Similar to the deployment described above, once the individuals on the team have validated the deployment, they might click a "promotion" button, or merge the changes into a master branch which will cause the application to be deployed to production.

You'll note that infrastructure is not described here at all. We're not using Chef, Puppet, or Ansible to maintain the environment, or Terraform to build new servers—we don't need to do so because the containers contain everything they need to run the application. There is no separate configuration or separate production build because the environment contains the variables the application needs to run. Service discovery also exists in the environment to allow applications to find the services they depend on.

Herein lies the real benefits of using a container orchestration framework instead of a traditional hypervisor with VMs—while the tools have more specialized abstractions, once set up, they require significantly less manual work to maintain and use. Developers can much more easily create integrations into the environment for work that they may be doing.

CHAPTER 5
Cloud Security

Attackers never start by attacking the component they're the most interested in exploiting. If an attacker is interested in stealing customer information from your database, they're unlikely to attack your database *first*. They're much more likely to get a foothold behind your firewall through a less-secure channel.

The two least-secure channels in enterprise systems are via *social engineering* and *known vulnerabilities*. We will cover both while focusing on the latter.

Your systems are only as secure as their weakest link, and the weakest link in an organization are the people themselves. It doesn't take much to convince a well-meaning person to give up critical information, including sensitive details such as passwords and other valuable information. Most people are hardwired to be helpful, and attackers exploit this weakness. Systems don't only need to defend against attackers from the outside, but also against attackers from the inside—even if the person themselves doesn't know that they're acting on behalf of an attacker.

In fact, we should assume that attackers are already in our systems, which is the cornerstone of NSA's defense in depth strategy.

> *There's no such thing as "secure" any more. The most sophisticated adversaries are going to go unnoticed on our networks. We have to build our systems on the assumption that adversaries will get in.*
>
> —Debora Plunkett, U.S. National Security Agency (NSA)

An attacker often uses strategic social engineering in tandem with exploiting known vulnerabilities in your software to gain an advantageous position. The first move is typically designed to establish a foothold within the system rather than attack it outright.

The application architecture we've outlined in this book helps teams to introduce *component isolation* through *bulkheading*. Bulkheads protect a ship from sinking if a single area of the hull is breached.

Like bulkheads in a ship, compartmentalization helps to contain breaches within a small area of the system. If a single service in a microservices-based system is compromised, there's a better chance of the intruder being contained within the compromised service rather than taking over our whole system—assuming proper security measures are in place.

Lines of Defense

The first critical line of defense is to arrange training for your people to identify and avoid common social engineering strategies. We highly recommend reading *The Art of Deception: Controlling the Human Element of Security* by Kevin Mitnick (Wiley).

From a technical perspective, applying the concepts from this book will give you the ability to *apply updates quickly*. While this is certainly a critical line of defense, there are many other implementation-level details that should be considered when modernizing legacy systems for the cloud.

"The IBM Secure Engineering Framework" suggests nine categories for security requirements. According to the SEF, "most application security vulnerabilities typically are caused by one of three problems":

- The requirements and design failed to include proper security.
- During implementation, vulnerabilities were inadvertently or purposefully introduced in the code.
- During deployment, a configuration setting did not match the requirements of the product on the deployment environment.

This chapter will focus on *the requirements and design failed to include proper security* problem, and specifically three critical security oversights in applications:

1. Applying updates quickly
2. Creating secure password hashes
3. Preventing the Confused Deputy

Applying Updates Quickly

One of the most common (and dangerous) security gaps is the lack of ability to rapidly patch known vulnerabilities in our tools and frameworks. Ironically, the most mission-critical systems in the world are often the most vulnerable.

Mission-critical systems must be available 24/7, so counter-intuitively they're the least likely to be up-to-date with the latest security patches. These systems are often legacy systems, which makes the risk of an outage due to unforeseen deployment issues nontrivial. The fear of prolonged outages is why some mission-critical software is so vulnerable.

This brings us to a catch-22:

- Systems that people depend on every day should be continuously up-to-date with the latest security patches.
- Mission-critical systems with legacy architecture means that applying patches requires downtime, so updates must be infrequent.

The only way to solve the root cause issue is to modernize mission-critical systems and adopt continuous integration and continuous delivery practices.

By following the architectural advice in this book we can modernize any type of legacy system, in turn helping to avoid downtime and maintenance windows during deployments. The improvements we introduce will eventually reduce the fear of keeping our systems up-to-date and secure against the latest known vulnerabilities.

Strong Passwords

In *Don't Build Death Star Security,* David outlined his recommendations for creating secure passwords that will be stored in a database. We will now cover how to create secure password hashes that are safe to store in a relational database.

NOTE

What's a Hash?

"A hash is designed to act as a one-way function: a mathematical operation that's easy to perform, but very difficult to reverse. Like other forms of encryption, it turns readable data into a scrambled cipher."[1]

Hashing is useless on its own. Attackers can easily determine all of the hashes in our database that are identical and run dictionary attacks against them, or compare the hashes in our database to hashes from other compromised databases. Hashing alone does virtually nothing for us, but it's the first step to create a *secure password hash*.

Hashes

We should never store passwords in plaintext; we should only store the *password hashes*. When a user creates a password, we hash the plaintext password they provide to us and store the hash in a database. When a user attempts to log in to the system, we compare the plaintext password they provide us with the hash in the database; if they match, the user is considered authenticated. If an attacker gains access to our database they will only see hashes, not plaintext passwords.

Salts

Hashes alone are not secure enough. We need to *salt* the hash. When a user creates a password, the plaintext password is added to the salt, hashed, and stored in the database. The salt is generated uniquely per password and stored in plaintext in the same row as the hash. The salt makes it impossible for an attacker to compromise *every* password in the database through dictionary attacks. For instance,

1 Andy Greenberg, "Hacker Lexicon: What Is Password Hashing?" Wired, 06.08.16.

without a salt, common passwords such as "123456" would all share the same hash, making it trivial for hackers to determine that repeating hashes are common (and likely easy to guess) passwords. So, without salts, attackers can attempt every known word in the dictionary as passwords, or passwords they've obtained from other exploits. To make brute force attacks even more expensive, it's critical that passwords are hashed using a strong algorithm such as PBKDF2 rather than a weaker hashing algorithm.

Peppers

A *pepper* is an additional layer of security, a secret combined with the plaintext password and salt but not stored in the database. The pepper is stored in encrypted form in a separate location using an SCM tool in Git or a secret service such as Vault. To create the secure password hash, we hash the plaintext password, the salt, and the pepper together. An attacker would need to compromise the secret store as well as the database to crack passwords in a stolen database. The main tradeoff of this technique is that users will not be able to log in to the system if the secret service is unavailable for any reason.

Password Stretching

To make it even more difficult to compromise an entire database worth of passwords, we need to significantly increase the computational power required to hash the password, salt, and pepper. Hashing requires computation, so by making computation more expensive we also increase the amount of time it takes to try different combinations. Imagine an attacker guesses at passwords one-by-one, and each guess takes *1 millisecond*. If we can increase each attempt to *1 second*, we've made guessing so time consuming that compromising even a single password is unlikely, let alone all passwords.

We increase the necessary computation required for hashing a password through *password stretching*. Instead of a single round of hashing, we perform many rounds of hashing—perhaps even hundreds of thousands of rounds. This will add overhead to a login service, but the extra delay during logon means that brute force attacks will take far too long to be a realistic attack vector. We must weigh the benefits of password stretching with the costs and tailor our use

accordingly, but often the cost is well worth the extra layer of security.

Assume Breach

Safeguarding passwords stored in SQL-based databases is one of the most critical aspects of security to consider, because compromised databases are one of the most common breaches.

We should always assume that an attacker has access to plaintext salts and hashed passwords in our database, and design our security implementation around those assumptions. Defense in depth requires us to assume that all components in our system can be—or already are—compromised. In the context of password hashing, the additional safety net of peppers along with password stretching provides enough security to limit cracked passwords to a single user rather than an entire database.

Preventing the Confused Deputy

Once we build distributed systems, one of the new vulnerabilities we're exposed to is the possibility of *impersonation attacks*. The Confused Deputy vulnerability is introduced once we move to a microservices architecture, and best summed up as "having internal systems that are too trusting."

Imagine we have a service that can perform a very privileged command, such as changing the password of all users. This would be a highly valuable service for a malicious hacker to compromise. As an attacker, the best way to compromise this service is to *impersonate* a user that has the authority to issue "update password" commands.

As we move towards more decoupled and distributed systems, it's often that our programs "take actions on the behalf of other programs or people. Therefore programs are deputies, and need appropriate permissions for their duties."[2]

The defense against impersonation attacks is to implement *capability-based systems*. Instead of *authenticating* a program or person and then *authorizing* them based on an internal list of capabilities they are allowed to perform, a *capability token* is provided that

2 Mark S. Miller, "Capability Myths Demolished", ERights, October 03, 1998.

validates the right for the *general* to issue the order to the *deputy*. This is a much finer grained method of security than identity-based authorization, such as ACL.

Rather than an administrator having an exhaustive list of all subjects that are able to carry out operations on other subjects, each subject itself knows what it can do, but it still must ask permission to do so before carrying out the order.

The capability model is in line with the *Principle of Least Authority* (POLA), which "relies on a user being able to invoke an instance, and grant it only that subset of authority it needs to carry out its proper duties."[3] As we mentioned above, this means that each instance which carries out actions (deputies) on behalf of others (generals) must have their own list of authorities, understanding which generals are authorized to issue which commands.

Conceptually, the workflow of capabilities-based security is straightforward. First, a capabilities system—perhaps along with simple public-key infrastructure (SPKI)—must be implemented. Once the system is in place, the command flow between two subjects is as follows:

1. The *general*, Alice, requests a *capability token* from the capability system.
2. Alice passes the token along with the command to her *deputy,* Bob.
3. Bob executes the instructions represented by the token, but only if the token is valid (Figure 5-1).

3 Mark S. Miller, "Capability Myths Demolished."

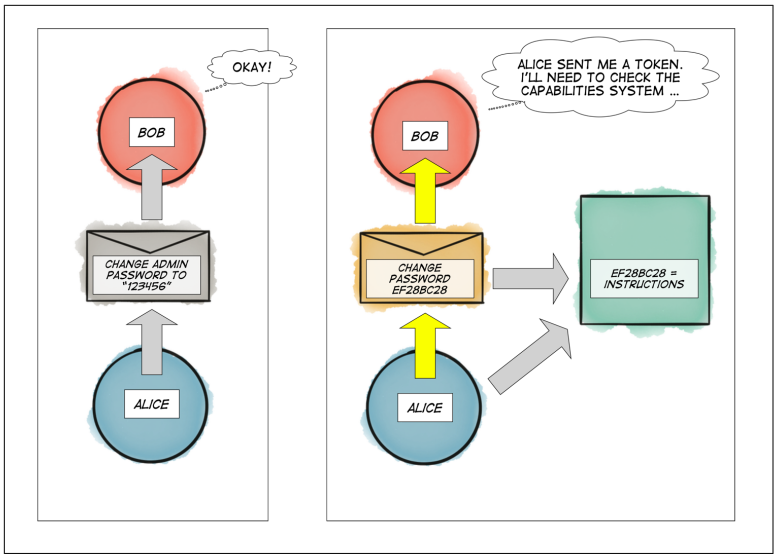

Figure 5-1. ACL-based security (left)—Bob thinks Alice is Alice and does as she commands. Capability-based security (right)—Bob accepts a token from Alice along with a copy of her key. Both Bob and Alice must be able to read from the capabilities system. Tokens are difficult to forge and valid only once.

With *access control list* (ACL)-based security, the services carrying out orders (the deputies) each have lists of "who can perform which action" embedded within themselves. If an attacker successfully impersonates a general it can instruct deputies as if they were the general themselves. It's conceivable that an imposter, "Malice," could impersonate Alice to trick Bob into carrying out treacherous orders. Impersonation attacks are much less likely to be successful when a trusted third party issues one-time-use capability tokens, which "bundles together designation and authority."[4]

NOTE

Man-in-the-Middle

The Confused Deputy exploit is similar to a *man-in-the-middle* attack where a malicious third party intercepts messages between two parties and alters those messages before they reach their intended destination.

4 Mark S. Miller, "Capability Myths Demolished."

Capability-based security is gaining wider recognition with the emergence of microservices. In fact, Google is working on a "capability-based, real-time operating system" called Google Fuchsia, which is rumored to fix some of the security issues currently present in Android. Companies such as Nuxi offer more practical implementations of capability systems for cloud computing, offering solutions such as CloudABI for "applying the principle of defense in depth to your software. It can be used to secure a wide variety of software, ranging from networked microservices written in Python to embedded programs written in C."

As we enter the era of the cloud, microservices, and IoT, we must continue to stay at the forefront of security. The architecture outlined in this book will help, but new attack surfaces will continue to be exposed and must be addressed. Defending a microservices-based architecture must never be treated as an afterthought. The defense in depth concept should be at the forefront of a reasonable modernization strategy in distributed systems, with the goal of providing "redundancy in the event a security control fails or a vulnerability is exploited that can cover aspects of personnel, procedural, technical and physical security for the duration of the system's life cycle." Regardless of your architecture, all software should assume breach and follow the defense in depth principles.

CHAPTER 6
Conclusion

This book outlines a new way of building applications for the next generation of cloud-native software, but this new approach has been decades in the making. Carl Hewitt, Peter Bishop, and Richard Steiger first published "A Universal Modular Actor Formalism for Artificial Intelligence" in 1973,[1] the basis of the actor model and the inspiration for much of what we covered in this book. Erlang—a programming language released over 30 years ago—has a message passing model heavily inspired by the actor model.

The cloud is actually one of the newest topics we've covered in this book, and it's quite young compared to actors and Erlang! Even append-only journals are a more traditional style of persisting state than relational databases. The relational model of data—eventually leading to relational databases—was first proposed by E.F. Codd in 1970,[2] barely more mature than the concept of actors. With virtually unlimited processing and storage now at our disposal, the habit of using relational databases for *everything* is coming to an end. It's time to rethink the way we build software for a new era of infrastructure. This is not the time to rest on our laurels. Cloud infrastructure, along with related JVM technologies such as Akka, will

1 Carl Hewitt, Peter Bishop, and Richard Steiger, "A Universal Modular Actor Formalism for Artificial Intelligence," *IJCAI*, 1973

2 E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, (1970) 13(6): 377–387.

usher in a wave of innovation in the development of business software.

Transitioning from traditional ways of building applications to building cloud-native systems is a major leap for all stakeholders in software projects, not only developers and operations teams, but every aspect of an organization who wishes to maintain a competitive advantage. Not only will languages, tools, and software development methodologies adapt to this changing world, but processes such as ITIL must be revisited and evaluated on their merits in a modern context. We strongly believe that while the material we have covered is challenging, learning this approach to software development will not only benefit the systems you build, but also enhance your career and keep you on the forefront of the software development industry.

> *I didn't have time to write a short letter, so I wrote a long one instead.*
> —Mark Twain

Don't be fooled by the short length of this book. We've covered the building blocks that will create a foundation towards building scalable, resilient, and maintainable software that will evolve with the needs of your business and stand the test of time. We hope this broad overview of modern development practices inspires you to learn new skills, reach new heights in your career, and thrive in this era of *the cloud*.

About the Authors

Kevin Webber runs a consultancy based out of Toronto, Canada that specializes in enterprise software modernization. Previously Kevin worked as an Enterprise Architect at Lightbend. He first applied many of the techniques discussed in this book on a modernization project for Walmart Canada, which delivered one of the first fully reactive ecommerce platforms in 2013. Kevin has over 17 years of enterprise software development experience; he started his career as a COBOL developer before transitioning to the world of Java and the JVM in 2001.

Jason Goodwin is Director of Engineering at FunnelCloud, where he is currently building an event-oriented manufacturing execution system. Prior to FunnelCloud, Jason was working with Google on a Scala-based video streaming platform. He also worked with Rogers Communications, helping to modernize their backend systems using some of the technologies and techniques outlined in this book.