THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES

MICROSERVICES WITH DOCKER ON MICROSOFT® AZURE

"Beyond just describing the basics, this book dives into best practices every aspiring microservices developer or architect should know."

—*Foreword by* **Corey Sanders,**
Partner Director of Program Management, Azure

BORIS SCHOLL
TRENT SWANSON
DANIEL FERNANDEZ

About This E-Book

EPUB is an open, industry-standard format for e-books. However, support for EPUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the e-book in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

Microservices with Docker on Microsoft Azure™

Boris Scholl
Trent Swanson
Daniel Fernandez

Addison-Wesley
Boston • Columbus • Indianapolis • New York • San Francisco
Amsterdam • Cape Town • Dubai • London • Madrid • Milan • Munich
Paris • Montreal • Toronto • Delhi • Mexico City • São Paulo • Sidney
Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2016937760

ISBN-13: 978-0-672-33749-9
ISBN-10: 0-672-33749-5

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing: June 2016

Editor-in-Chief
Greg Wiegand

Acquisitions Editor
Trina MacDonald

Marketing Manager
Stephane Nakib

Development Editor
Mark Renfrow

Managing Editor
Sandra Schroeder

Technical Reviewers

Marc Mercuri
Nigel Poulton

Project Editor
Lori Lyons

Project Manager
Ellora Sengupta

Copy Editor
Abigail Manheim Bass

Indexer
Lisa Stumpf

Proofreader
Suriyanarayanan

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasersith

Compositor
codeMantra

To my lovely wife Christina and to my children Anton and Marie, who are the joy of my life and make every day so special. Thanks for being there for me every day.
—Boris

To my wife Lisa and my son Mark who gave up a lot of evenings and weekends while I was writing this book, my parents who taught me the lesson of hard work and perseverance, and all 12 of my brothers and sisters for pretending to care. Thanks for your support while writing this book.
—Trent

My immense gratitude to mother and father who supported my interest in computers from a young age. To my wife and best friend Angie, my son Gavin, and daughter Divina who are my reason for being. Thank you for all your support while writing this book.
—Dan

Contents at a Glance

Foreword
Preface
Acknowledgments
About the Authors

1 Microservices

2 Containers on Azure Basics

3 Designing the Application

4 Setting Up Your Development Environment

5 Service Orchestration and Connectivity

6 DevOps and Continuous Delivery

7 Monitoring

8 Azure Service Fabric

A ASP.NET Core 1.0 and Microservices

Index

Contents

Foreword
Preface
Acknowledgments
About the Authors

1 Microservices
- What are Microservices?
  - Autonomous Services
  - Small Services
- Benefits of Microservices
  - Independent Deployments
  - Continuous Innovation
  - Improved Scale and Resource Utilization
  - Technology Diversity
  - Small Focused Teams
  - Fault Isolation
- Challenges
  - Complexity
  - Network Congestion and Latency
  - Data Consistency
  - Testing
  - Integration and Versioning
  - Service Discovery and Routing
  - Monitoring and Logging
  - Skillset and Experience
  - Uptime Service Level Agreement
- Best Practices
  - Encapsulation
  - DevOps Principles and Culture
  - Automation
  - Monitoring
  - Fault Tolerance
- Summary

2 Containers on Azure Basics
- VMs, Containers, and Processes

When Would We Use a Container Over a Virtual Machine or a Process?
Containers on Azure
Creating an Azure VM with Docker
Generating an SSH Public Key on Windows
Generating an SSH Public Key on Mac OS X
Choosing a Virtual Machine Image
Connecting to the VM Using SSH and Git Bash on Windows
Connecting to the VM Using SSH and Git Bash on Mac OS X
Docker Container Basics
Summary
3 Designing the Application
Determining Where to Start
Coarse-Grained Services
Starting with Microservices
Defining Services and Interfaces
Decomposing the Application
Service Design
Service to Service Communication
Synchronous Request/Response
Asynchronous Messaging
Monolith to Microservices
Flak.io e-Commerce Sample
Flak.io
Requirements
Architecture Overview
Considerations
Summary
4 Setting Up Your Development Environment
Using Docker for Local Development
Docker for Local Development
Docker for Production Validation
Docker as a Build/Test Host
Developer Configurations
Local Development
Local and Cloud
Cloud Only

Managing Docker Authentication
Choosing a Base Image
Build a Hierarchy of Images
Setting up your Local Dev Environment
Install Docker Tools
Install Developer Tools
Install Windows Utilities
Install OSX Utilities
Docker for Local Development
Local Development Settings
Starting your Local Docker Host
Connecting to a Docker Host
Cloning Samples
Enabling Live Reload in a Docker Container
Volumes
Preparing your Microservice for Production
Docker Compose
Debugging Docker Issues
Unable to Connect to the Docker Host
Containers That Won't Start
Diagnosing a Running Container
Summary
5 Service Orchestration and Connectivity
Orchestration
Provisioning
Infrastructure as Code
Azure Resource Manager
Azure Container Service
Multivendor Provisioning
Scheduling and Cluster Management
Challenges
A Scheduling Solution
Docker Swarm
Kubernetes
Apache Mesos
Using Apache Mesos to Run Diverse Workloads

Service Discovery
Service Registration
Service Lookup
Service Registry
Technologies
Other Technologies
Application/API Gateway
Overlay Networking
Summary
6 DevOps and Continuous Delivery
DevOps Overview
Modern DevOps
DevOps Culture
Continuous Integration, Delivery, and Deployment
Creating Environments in Azure
Deploying a Microservice with Continuous Delivery
Application Configuration Changes Across Different Environments
Continuous Integration
Testing in a QA Environment
Deploying to Staging
Testing in Production
Choosing a Continuous Delivery Tool
On-Premises or Hosted?
On-Premises or Hosted Build Agents?
Best-of-breed or Integrated Solution?
Does the Tool Provide the Extensibility You Need?
Comparison of Jenkins, Team Services, Bamboo, and Tutum
Docker Cloud (Formerly Called Tutum)
Summary
7 Monitoring
Monitoring the Host Machine
Monitoring Containers
Monitoring Services
Monitoring Solutions
Azure Diagnostics
Application Insights

Operations Management Suite (OMS)
Recommended Solutions by Docker
Summary
8 Azure Service Fabric
Service Fabric Overview
Service Fabric Subsystems
Cluster Management
Resource Scheduling
Service Fabric Application
Custom Applications (Existing Applications)
Container Integration
Service Discovery
Programming Model
Stateless Services
Stateful Services
Reliable Actors
Reliable Services
Application Lifecycle
Service Updates
Application Upgrades
Testability Framework
Summary
A ASP.NET Core 1.0 and Microservices
A New Version of ASP.NET
Getting Started
Choosing the Right ASP.NET Docker Image
Visual Studio 2015 Tooling
ASP.NET Microservices Best Practices
Index

Foreword

Over the last couple of years, we have seen Azure evolve from a simple .NET-based platform to an open and flexible platform, supporting the broadest selection of operating systems, programming languages, frameworks, tools, databases and devices for infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS), and software-as-a-service (SaaS) workloads. As a result, Azure is growing at an amazing rate with both existing and new customers.

Today, there is not a single industry that does not consider making use of the cloud in one form or another, from big compute to Dev/Test to SaaS solutions. For IT and developers, flexibility and agility are the number one reason for adopting Azure. A typical pattern of customers adopting Azure is to start with dev/test scenarios, followed by moving existing applications to run IaaS-based hybrid scenarios, and eventually developing new applications to take full advantage of the cloud platform.

The Azure cloud infrastructure is now in a place where it provides the flexibility to accommodate almost every scenario. Thus, customers have realized that their application design is now the limiting factor. Many customers still have a monolithic application design in place that makes it hard to independently update, version, deploy, and scale individual application components. Therefore, despite the cloud being extremely agile and flexible, the application itself limits the agility needed to react quickly to market trends and customer demands.

Over the last couple of months, microservices-based applications have become the most talked-about new architectural design, enabling previously impossible agility and ease of management. Docker containers turn out to be a perfect technology to enable microservice-based applications, from a density, DevOps, and open technology perspective. When coupled with Docker, microservices-based applications are a game changer when it comes to modern application development in the cloud.

I'm really excited that Azure offers the foundational technology and higher-level services to support any type of microservices-based application. You can build applications using Docker containers on Apache Mesos with Marathon/Chronos/Swarm or you can build applications on our own native microservices application platform, Service Fabric. Azure offers the right choice for your scenario.

Whether you have just gotten your feet wet with containers or microservices, or you have already advanced in that subject, this book will help you understand how to build containerized microservices-based applications on Azure. Beyond just describing the basics, this book dives into some of the best practices each aspiring microservices developer or architect should know. Boris, Trent, and Dan are the very best people to walk through both the basics and the advanced topics! All of them have deep real-world experience building applications using these models, and amazing product insight on Azure and the cloud. I am excited to see what you can build using the skills they share in these pages!

—Corey Sanders
Partner Director of Program Management, Azure

Preface

The three of us have been working with the Microsoft Azure Cloud platform since its first release in 2009. Our collective work with the platform is fairly broad—from building the platform and applications on the platform to creating the Azure development tools and experiences. In addition, we have enabled numerous customers and partners to build their large-scale cloud-native applications on Microsoft Azure. Over the years, we've been able to apply many lessons learned, ranging from designing applications for resiliency and scale all the way to DevOps best practices from our interactions with customers to Azure platform capabilities, Azure tooling, and technical documentation.

However, some questions and problems continued to persist. Just to name a few, how do we make sure that what works on my development machine also works in a cloud environment? How should we think about structuring the application so that we do not need to update everything if there is just a minor change to one component? How do we deploy updates as fast as possible without downtime? Finally, how do we handle configuration and environment changes?

In 2013, we began hearing more industry leaders and customers talk about Netflix, Amazon, and other businesses using microservices as an architectural approach to address those challenges. We did a head-to-head comparison with our most successful architectures (with both internal and external customers) and realized that we had already implemented many characteristics of microservices patterns—for example, designing Cloud Services applications based on workloads or the decomposition of an application into multiple services with the lifecycle of individual components/services as the motivation. Clearly, architectures were evolving in this direction, and when the term "microservices" became popular, many architects and developers realized theirs were already heading in that direction.

Enter Docker. Docker reduces deployment friction and the cost of placing a single service into a single host. This reduced deployment friction helped manage the deployment of the growing number of services common in a microservices architecture and helped standardize the deployment mechanisms in a polyglot environment. The programmable infrastructure offered by cloud environments, along with containers, paved the way for microservices architectures.

But having the right architectural approach and tool is just half the equation. Conceptually thinking about how to set up development and test environments, automate the DevOps flow, orchestrate and schedule Docker containers in a cluster of virtual machines, how to make the microservices discoverable by other services, and how to monitor the environments and the services, constitute the critical other half.

We, the author team, have spent the last two years working on microservices and Docker scenarios on either the Visual Studio tooling for the Docker engineering team, the Azure Service Fabric Compute engineering team, or working directly with our customers.

We wrote this book to share the hard-earned lessons we've learned and provide you with the tools you need to succeed building microservices with Docker on Azure.

About This Book

This book provides an insider view of many of the services and features mentioned throughout the book. We participated in the design and implementation of many of these features as well as working

with many internal teams, such as Azure DB, Skype, and Cortana, who have successfully been using microservices architectures for years. Therefore, we can speak from a unique perspective when it comes to the considerations and challenges one has to take into account for designing and building microservices-based applications.

This book is aimed at anyone who is interested in building microservices-based applications on Azure. After reading this book, you will have a solid understanding of both, the benefits and the challenges of microservices-based applications. You will gain knowledge that you can apply to design microservices-based applications on Azure either from ground up or to decompose an existing monolith application into microservices over time.

Readers will come away with the following:

- An understanding of the difference between microservices-based applications and traditional monolith applications, and the pros and cons for each approach.
- An understanding of Docker containers in the context of microservices architectures and basic Docker operations, as well as how to create Docker hosts on Azure.
- Best practices for setting up development and DevOps environments for microservices-based applications.
- An understanding of cluster and container orchestration capabilities on Azure.
- Best practices for monitoring containerized microservices applications and monitoring tools available on Azure.
- An understanding of Azure Service Fabric and how it enables developers to develop microservices-based applications.

Docker Coverage in This Book

The Docker ecosystem is growing at an amazing rate. At DockerCon 2015, Docker reported that between 2014 and 2015, the number of pulls from the Docker Hub increased 18,000% to more than 500 million! New features were added daily by Docker and its respective community. This book is about microservices with Docker on Azure, so we cover Docker tools and features available on Azure throughout the book. While it's impossible to fully cover everything the Docker ecosystems provides, our goal is to provide a solid overview of the most useful Docker services and features available on Azure.

At a high level, Azure's support and strategic direction for Docker is about enabling your choices:

- **Choose your development tool**—Whether that's Visual Studio, Eclipse, Atlassian, Jenkins, Docker Hub Enterprise, or simple command-line tools, if it works with Docker, it will work with Azure.
- **Select your provisioning or configuration tools**—Choose from Puppet, Chef, Docker Machine, Azure Resource Manager, PowerShell, cross-platform CLI, or via the Azure website portal.
- **Pick your clustering tool**—Azure includes support for the Azure Container Service as Platform-as-a-Service for Docker based on Mesosphere, as well as the ability to run your own clustering solution like Kubernetes, DCOS, Docker Swarm, and more.
- **Work with the Operating System of your choice**—Windows Server 2016 and many Linux operating systems are supported, including Ubuntu, CoreOS, and Red Hat Enterprise.

- **Enable Public or Private Cloud**—Azure provides both a public cloud as well as a private hosted cloud for government, and with Azure Stack, enterprises will be able to host a privately hosted, on-premise cloud inside your firewall.

In order to build and deploy containerized services we need an infrastructure for development and test production. Azure offers a set of fundamental services to build, ship, and run Docker.

Azure Virtual Machines as a Docker Host

The foundation for every Docker environment in Azure are Azure virtual machines (VMs). There are several ways to create a Docker host, that is, a virtual machine installed with Docker. In order to support running Docker, Azure provides a Linux virtual machine extension that installs both Docker and Docker Compose. While Windows Server 2016 has not yet officially released, it too will have an intuitive way to provision a Docker-ready virtual machine. Below are sections of the book that go into detail about setting up a Docker host in Azure:

- Chapter 2
  - Docker VM extension
  - Create a Docker host through the Azure portal
  - Create a Docker host through Azure Resource Manager
  - Create a Docker host in Azure through command line interface (CLI)
  - Create a Docker host in Azure through docker-machine
  - Setting up a Docker host on a local machine using Docker toolbox
- Chapter 4
  - Setting up a Docker host on a local machine
- Chapter 5
  - Azure VM extensions
  - Create a Docker host through Azure Resource Manager

Cluster Management and Orchestration on Azure

To run, operate, and scale containerized microservices applications in a resilient way, we need a collection of Docker hosts called a cluster to make sure that there are always enough Docker hosts available even in the case of failures. In addition, we'll need a scheduler (also often referred to as orchestrator) to place the containers in Docker hosts across the cluster. Azure offers great flexibility when it comes to cluster deployment and management as customers have the choice of what clustering infrastructure to run. The list below highlights chapters that cover those areas.

- Chapter 5
  - Cluster management on Azure with Kubernetes, Mesosphere DCOS, and Docker Swarm
  - Setting up a Mesos cluster with Marathon and Chronos using the Azure Container Service
- Chapter 8
  - Azure Service Fabric cluster management and orchestration capabilities

Development and DevOps for Docker-Based Microservices Applications

DevOps, including monitoring, play an important part in microservices development. Microsoft offers a variety of tools that help with development, DevOps, and monitoring. Given the breadth of the topic, we split it into four chapters. Below is a list of what is covered in each chapter.

- Chapter 4
  - Docker support in Visual Studio code
  - Options for running a Docker registry service in Azure
- Chapter 6
  - Continuous Delivery options for Docker
- Chapter 7
  - Monitoring Docker-based microservices applications using Azure Diagnostics, Visual Studio Application Insights, and Microsoft Operations Management Suite
- Chapter 8
  - Overview of developing and managing microservices in Azure Service Fabric

Working with Docker

The list of Docker features and commands is constantly growing. Starting with Chapter 2, we explain the most common Docker features and commands that enable you to successfully build and use containers in a microservices-based application.

- Chapter 2
  - Pulling Docker images from Docker Hub
  - Building Docker images
  - Docker image layering
  - Docker volumes
  - Working with containers
  - Linking containers
  - Docker network capabilities
  - Environment variables
- Chapter 4
  - Docker authentication
  - Docker images and tags
  - Docker containers and source code management
  - Docker volumes
  - Docker compose
  - Docker logs
- Chapter 5
  - Docker Swarm
  - Docker Compose
  - Docker Networks

- Chapter 6
  - Docker Tutum
- Chapter 7
  - Monitoring Docker-based microservices applications using Azure Diagnostics, Visual Studio Application Insights, and Microsoft Operations Management Suite

Code Samples and Software Requirements

Good explanations come with effective code examples. To truly experience key patterns of microservices architectures, we built an extensive sample application to demonstrate microservices concepts discussed throughout the book. The individual services are written in different languages (Node.js, Go, ASP.NET5) using different data stores (mySQL, Elasticsearch, block storage). The infrastructure uses Docker hub as a service registry and the Microsoft Azure Container service for cluster management and orchestration of the services.

The sample application is available on GitHub at https://github.com/flakio/flakio.github.io.

We hope you enjoy building containerized microservices applications as much as we enjoyed writing about them.

—Boris, Trent, Dan

Acknowledgments

Special thanks to Bin Du for his help with the Monitoring chapter, particularly the Azure diagnostics part. Bin Du is a senior software engineer in Microsoft who has been working on Visual Studio diagnostics tools for Azure platform for more than five years. His experience and insights in diagnostic telemetry, performance troubleshooting, and big data analytics made it possible to write about Azure diagnostics and diagnostics practices from an inside view of a developer who is working in those areas.

Special thanks also to Den Delimarschi for helping author the Continuous Delivery chapter. Den is a Program Manager who had previously worked on improving the build and continuous integration process for Outlook so his insights and hands-on experience were invaluable.

This book would not have been possible without the help of many people. Special thanks to all of the reviewers for their thoughtful comments and discussion, which has helped to make the book of such a great quality. In particular, thanks to Ahmet Alp Balkan, Bin Du, Chris Patterson, Christopher Bennage, Corey Sanders, Den Delimarschi, Donovan Brown, Jeffrey Richter, John Gossman, Lubo Birov, Marc Mercuri, Mark Fussell, Masashi Narumoto, Matt Snider, Vaclav Turecek.

We also thank our families and friends who have contributed in countless nontechnical ways. We could not have done it without your support!

About the Authors

Boris Scholl is a Principal Program Manager on the Microsoft Azure compute team, looks after Service Fabric custom application orchestration, container integration, and Azure's OSS developer and DevOps story for container based workloads. Prior to this, he was leading the Visual Studio Cloud Tools team focusing on architectural and implementation patterns for large-scale distributed Cloud applications, IaaS developer tooling, provisioning of Cloud environments, and the entire ALM lifecycle. Boris gained his experience by working as an architect for global cloud and SharePoint solutions with Microsoft Services. In addition to being a speaker at various events, Boris is the author of many articles related to Azure development and diagnosing cloud applications, as well as co-author of the book *SharePoint 2010 Development with Visual Studio 2010* (Addison Wesley Professional ISBN 10 0321718313).

Trent Swanson is a typical entrepreneur. As a cofounder and consultant with Full Scale 180, he works with some of Microsoft's largest customers, helping them migrate and build applications on the Microsoft Azure platform. He has been involved in building some of the largest applications running on Microsoft Azure today, some of which now utilize Docker and a microservices architecture. Trent often works with the Microsoft Patterns and Practices team developing guidance and best practices for cloud applications, where he also co-authored a book on cloud design patterns. As a cofounder of Krillan and Threadsoft, he has built applications based on a microservices architectural style using Docker, Node.js, Go, and Mesos. As a cofounder of B & S Enterprises, he dabbles with various IoT technologies for commercial building management.

Dan Fernandez is a Principal Director managing the Developer Content teams for Visual Studio, Team Services, ASP.NET, and parts of Azure. Prior to this, Dan worked as a Principal Program Manager managing the developer experience for Docker including Visual Studio, Visual Studio Code, and Docker continuous integration using Visual Studio Team Services. Dan is also the author of the Channel 9 video series Docker for .NET Developers. You can find Dan on Twitter at @danielfe.

Tech Editors

Marc Mercuri is currently a Principal Program Manager in the Azure Customer Advisory Team (CAT) and the functional lead of the team's strategic accounts program. Marc has been actively working in the software and services industry for the past 20 years. The majority of this time has been focused on distributed computing scenarios, with the past 6 years exclusively focused on cloud-centric (PaaS, SaaS, and IaaS) enterprise projects and commercial products.

Marc has been involved in the architecture of hundreds of high-profile, high-scale public and hybrid cloud solutions. The result is significant experience in designing, implementing, operating, and supporting reliable cloud solutions and services at scale. This experience includes both general cloud architecture best practices as well as nuanced considerations for key scenarios (IoT, Enterprise Hybrid, Consumer Mobile, Big Data Pipelines, Machine Learning, Deployment Automation, Open Source Software Deployment/Configuration, etc.).

Marc is the author of four books on services and identity and is a speaker at internal and external conferences. He has 10 issued patents and 17 patents pending in the areas of cloud, mobile, and social.

Nigel Poulton is a self-confessed tech addict who is currently living it large in the container world.

He creates the best IT training videos in the world for Pluralsight (those are his words) and co-hosts the In Tech We Trust Podcast. In his spare time he does a good impression of a husband and a father. He lives on Twitter @nigelpoulton and he blogs at http://nigelpoulton.com.

1. Microservices

Software is consuming the world and becoming a core competency of nearly all businesses today, including traditional businesses whose focus is not specifically technical. A growing number of industries need software to remain relevant, and that software must be able to evolve at a rapid rate to meet the demanding needs of today's fast-changing and competitive markets. The way we build and manage software continues to evolve as new technologies enter the market and we learn of new ways to build better systems. Many early concepts still hold true although others have simply evolved, or have been adapted to new technologies. New applications are showing up every day to meet the growing needs of consumers and businesses alike.

As we begin building a new application, the team and codebase are generally small and agile when compared to a more mature application. As new features are added to the application, the team and codebase continue to grow. The increased size of the team and codebase brings some new challenges and overhead that can begin to slow progress. The code becomes increasingly difficult to understand and reason about, leading to longer development ramp-up times. Enforcement of modularity becomes challenging, the encapsulation easily breaks down, and the application grows increasingly more complex and brittle, requiring additional coordination across the teams when changing features. Builds and testing begin to take longer and releases become more fragile, including a larger set of changes with each deployment. The smallest change to a component can require the redeployment of the application, which can take a long time to build, test, and deploy. If one component of the application needs to be scaled, the entire application needs to be scaled. Everything about the application is tightly coupled: teams, builds, deployments. This might not be a problem for some applications, but for large-scale applications that must evolve at a rapid rate, this can be a big problem.

For a long time now we have understood the benefits of componentization and encapsulation in software, but enforcement can be challenging. Through the recent DevOps shift we now understand the benefits of smaller incremental releases, and making the entire team accountable for the complete lifecycle of the application. When building large-scale applications, we understand the benefits of partitioning, and the cost of coordination on scale. As with applications, some organizations can also benefit from breaking down into smaller teams, helping to reduce coordination overhead.

Over recent years many organizations have begun to realize the benefits of decomposing applications into smaller isolated units that are easier to deploy, and structuring teams around those units. This decomposition of applications into smaller autonomous units is part of what is called a microservice architecture.

What are Microservices?

Microservices is a term used to refer to a service-oriented architecture, in which an application is composed of small autonomous services. These services are very loosely coupled, small (micro), and focused on a single feature of the application.

> **Note**
>
> A microservices architecture has also been referred to as "fine-grained SOA." We will not discuss the differences and similarities between a microservices architecture and

service-oriented architecture (SOA) in this book. Both are service architectures dealing with distributed systems communicating over the network. A microservices architecture tends to have a specific focus on incremental evolution over reusability.

Instead of building one single codebase all developers touch, which is often large and risky to manage, the application is composed of smaller codebases independently managed by small agile teams. The only dependency the code bases have on one another is through APIs (application-programmer interfaces). This architectural style has become popular with large-scale web companies like Netflix, eBay, and Amazon. These companies were struggling to scale their applications and, at the same time, bring new features to market rapidly with their existing monolithic architecture. They realized that instead of building one monolithic application, they would build smaller services to handle discrete business functions. The services are self-contained, and the boundaries between the services are well-defined APIs, exposing the business capabilities of the service.

In Figure 1.1, we see a monolithic application with all features and components of the application built and deployed together as a single unit. The monolithic application can be scaled out to multiple instances, but it's still deployed as a single application.

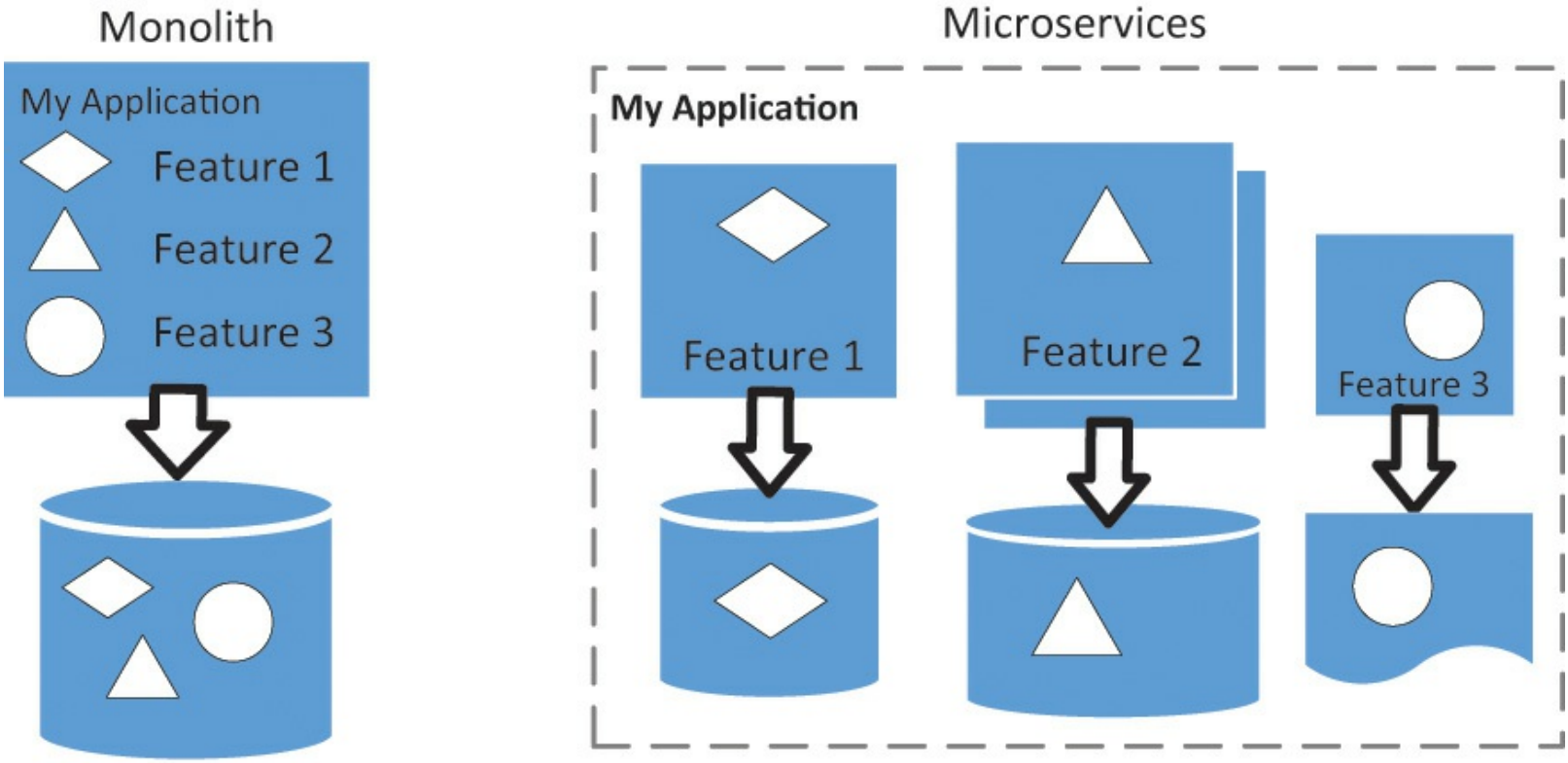

FIGURE 1.1: A monolith and a microservices high-level structure

A microservices implementation of the application places each of the features in separate services that are built and deployed independently of the other services. Together the independent services provide the functionality of "My Application." In an ecommerce application this could be the shopping cart, catalog, search, recommendations, account management, checkout, and so on. Team members working on recommendations would not need to concern themselves with all the details of managing shopping carts and purchases. They would likely need information from these services to make good recommendations, but not all the other details for handling the purchase. Each service could use the technologies best suited for each service and can be scaled independently as necessary for the individual service.

Flak.io ecommerce Sample

The flak.io microservices sample code uses an ecommerce scenario to demonstrate the concepts discussed in this book. The sample code can be found on GitHub at (https://github.com/flakio).

There are a number defining characteristics commonly present in a microservices architecture. Many of the characteristics that been used to describe microservices are generally in support of loose coupling and small services.

Autonomous Services

Microservices is an architecture that is optimized around autonomy. Autonomy and loose coupling between the services are important characteristics of a microservices architecture. Loose coupling means that we can independently update one service without making changes to another. The loose coupling extends throughout the entire service, across the teams, and right down to the data store—a type of coupling that is often overlooked is at the data store.

Microservices interact through APIs, and they do not share database schema, data structures, or internal representations of the data. Teams that have been building web applications might be familiar with some of these concepts and have consumed other third-party services when building their application. If we are consuming a service from some third-party geocoding service, we would not integrate through the database or a shared service bus that couples our system to the third-party service. Instead, we would integrate through an API, and that API would be well documented, versioned, and easy to consume. If the service owner needs to fix a bug in the API, they can make and deploy the change without coordinating this effort with all the consumers of the API. These changes should not affect consumers of the application and if for some reason a breaking change is needed, then a new version of the interface is deployed and consumers can move to this version when they are ready.

You can think of each service as its very own application, having its own team, testing, build, database, and deployments. These services are built and managed like any other small application. They must remain independently deployable and will generally service a single purpose. This brings us to the next topic, the size of the individual services.

Small Services

The individual services in a microservices architecture generally adhere to the single responsibility principal. The services are meant to be simple, doing one thing and doing it well. In theory if a service has more than one responsibility then we should consider breaking it up in to separate services.

Single Responsibility Principle

The single responsibility principle simply states that every class in an object-oriented programming language should be encapsulated and responsible for only a single piece of functionality.

In reality, following this principle might not make sense at times, and there can actually be good

reasons to keep some of the services more coarsely grained. Some functions of the system may inherently be more tightly coupled or it may be too costly to decompose. This could be due to the need for transactional consistency, or it may just be that the features are highly cohesive and become too chatty.

The question still remains as to how big or small a service must be for it to be a properly sized microservice. There are no specific rules to measure the size of a service, but there are some general guidelines the microservices community has identified, which work well in most situations. Many start with Amazon's principle that a service should be no bigger than one that can be developed and managed by a two-pizza team (the entire team can be fed with two pizzas). Others will use lines of code or the ability for team members to understand the entire service. Whichever metrics are used, the goal is that the individual services have a single purpose and can be easily understood and managed by small teams. As engineers, we will need to be pragmatic and consider various factors when defining service granularity and boundaries, like cohesion, coupling, communications, and data architecture.

Defining service boundaries can be one of the most challenging tasks with a microservices architecture, and doing so requires a strong understanding of the business domain. Whether decomposing an existing monolith or starting a new application, we start from the business perspective, considering the business capabilities. A good approach to identify the boundaries that separate services is using a bounded context from domain-driven design (DDD). We will cover more of the details for defining service boundaries in Chapter 3, "Microservices Design."

Domain Driven Design (DDD)

Domain Driven Design (DDD) is a set of principles and practices to help developers define business domain models as software abstractions. We can use this process to help identify boundaries and break down our application in to individual microservices, based on business function.

Benefits of Microservices

Now that we know a bit of what these microservices are about, let's have a look at the benefits provided through this approach and why we might consider using this strategy in our application.

Independent Deployments

With a large monolithic application, fast reliable deployments can be problematic. When updating a feature or fixing a bug in a monolithic application, we typically need to build and deploy the entire application. Even the smallest change to a library or component will require a redeployment of the entire application, including all those other components that have not changed. Depending on the size, technologies, and processes used, building and deploying an update can take a significant amount of time. In some situations it can take many hours to push out a new build for even small changes or bug fixes. Not only is our update potentially held up by other changes that need to be deployed with it, those other changes may cause deployment failures and require a rollback of the deployment. If we could decouple changes to one feature of the application from others, these problems could be avoided.

As a monolithic application grows, so does the fear of making changes, and as a result, all

development slows down. Businesses and IT departments avoid touching large monolithic applications due to the increased risk of failure and long roll-back processes. This results in a stalemate situation where an update is required but is not implemented due to the perceived risks. By decoupling changes to one component of the application from other components, only the parts that changed need to be released.

In a microservices-based architecture, each microservice is built and deployed independently of the other services, provided we are able to maintain loose coupling between the services. This means that we can deploy a critical bug fix to our payment processing component (or microservice) completely independently of other changes needed by the catalog service. If the changes to the catalog need to be rolled back, they can be rolled back while our payment processing bug fix remains in production. This makes it possible for large applications to remain agile, and to deploy updates more quickly and more often. A microservices design removes the coordination overhead, as well as the fear factor of making changes. Quite often change advisory boards are no longer needed and changes happen far more quickly and efficiently.

> **Note**
>
> Fast, reliable deployment through automation is an important factor to reducing Mean Time to Resolution (MTTR).

Not only do the smaller services enable these separate deployments and updates, given that we only have to build a small part of the application, build times can be much faster. Depending on the size of the application, this can be significant, even with parallel builds and tests.

Continuous Innovation

Everything in business and technology is changing at a rapid pace, and companies can no longer seek and rely on sustainable competitive advantage. Companies instead need to innovate continuously to stay competitive, or even just to remain in business. Organizations must remain agile and able to respond to fast-changing market conditions very quickly, and software has to be able to quickly adapt to these changing business needs.

It's not uncommon for organizations to refrain from pushing updates during their most busy seasons, unless they absolutely have to. For example, ecommerce sites have generally stopped deploying noncritical updates to systems during the busy holiday season. The thinking is, "Don't touch anything unless we absolutely need to because we dare not break something right now." Downtime during these critical periods can be extremely costly to the business—even a few minutes could mean significant losses in sales.

Through independent deployments and rigorous DevOps practices, a microservices architecture can make it possible for organizations to release updates during these critical times without adversely impacting the business. The software and services can continue to evolve, and valuable feedback can be gathered, enabling a business to innovate at a very rapid rate.

For example, during the holiday season a new recommendation service can be deployed without affecting the rest of the application. A/B testing can be performed during these times of heavy usage, and knowledge gained in this way can be used to improve conversions, or the user experience in the application. Fixes and new features can be deployed using release management strategies, like blue-

green deployments and canary releases. If there is a problem with the changes, they can be quickly rolled back. Even if the updates temporarily break recommendations or performance is a problem, the components handling the purchase of goods remain untouched. Even if we break something, a good deployment strategy would impact a small number of customers, the problem would be detected early, and we would roll back. Deployments in these situations cease to become an event, and are simply daily routine. See Chapter 6, "DevOps and Continuous Delivery," in this book for more information on microservices release management strategies.

The capability for a business to continuously innovate in today's competitive markets is often reason enough to consider a microservices architecture approach.

Improved Scale and Resource Utilization

Applications today are commonly scaled up by increasing the resources available to one or more deployments of the application, or they are scaled out by increasing the number of instances. Basically, we either put our application on bigger boxes, or create more instances of the application, and then load balance across those instances. In the case of a monolith, if one feature of the application requires additional scale, the application as a whole is scaled up or out.

We can, for example, have a feature that requires a lot of memory to perform its work, and only a few number of instances. Ideally this task would run on a few high memory instances. Another feature in the application might use more CPU or bandwidth and run optimally on a larger number of smaller instances. Because the features are compiled into a single application, we need to deploy the application to resources that satisfy both of these requirements. As the application scales, we may end up with a lot of unused, costly resources using instances that must meet these very different resource requirements.

A microservices architecture allows us to scale features of the application as needed and deploy the features to instances that better match their resource requirements. If the catalog service needs to be scaled up to meet demand, it can be without having to scale the other services that compose the application. If the recommendation service requires a lot of memory, it can simply be deployed to instances with more memory. It's still possible to deploy the various services to the same instances, but we have the ability to better optimize for cost and scale in our deployments. In Figure 1.2, we can see that Service A is deployed to a small number of large instances whereas Service C is deployed to a larger number of smaller instances.

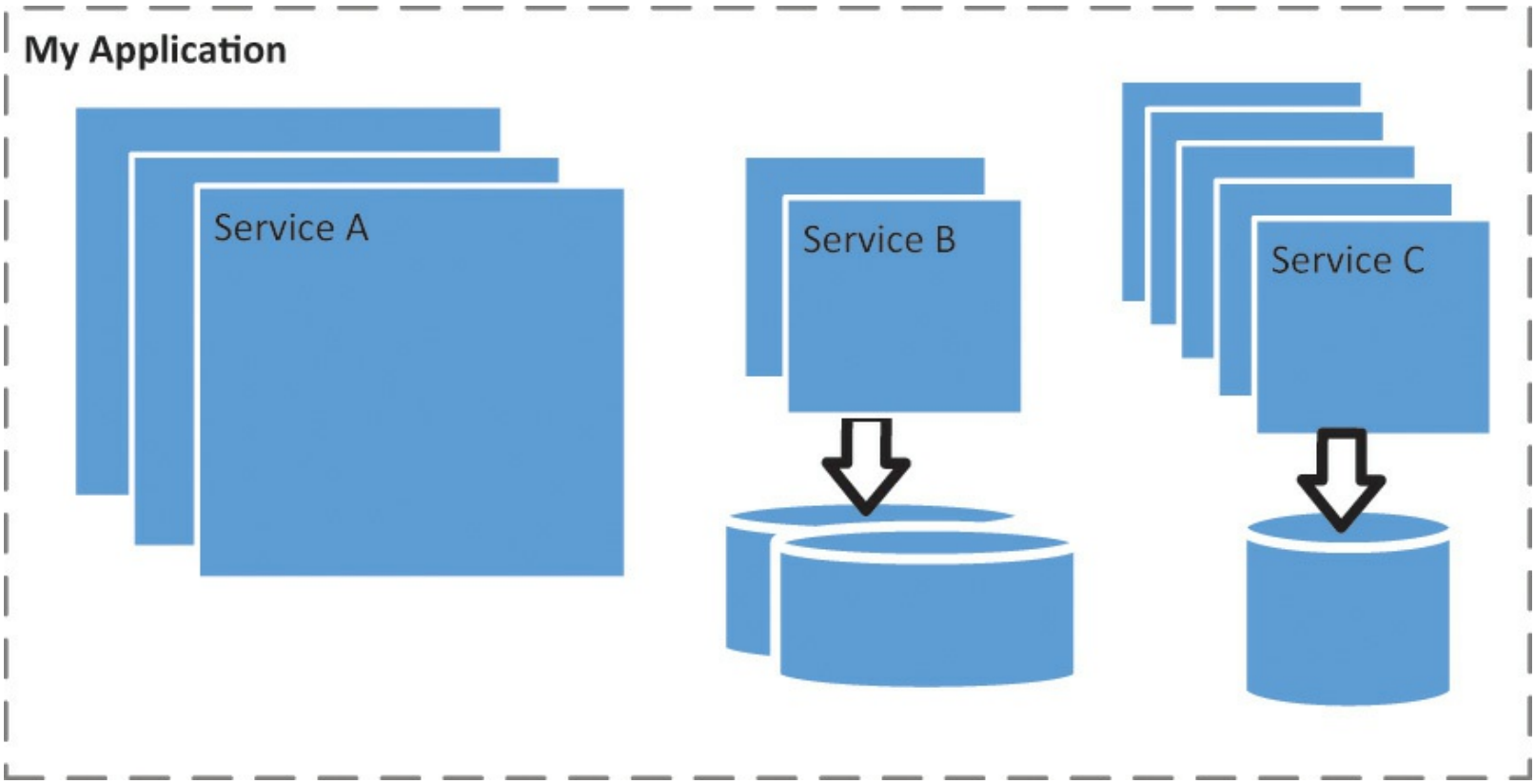

FIGURE 1.2: Different sizes and instances of services

Technology Diversity

In the monolithic application, everyone needs to agree on a language, stack, and often a specific version of a stack. If one component of an application can benefit from some new features in the most recent .NET framework release, we may not be able to adopt it because another component in the application might be unable to support it.

Unlike the libraries and components of a monolithic application, with a microservices approach the application can be composed of independent services that leverage different frameworks, versions of libraries or frameworks, and even completely different OS platforms.

Technology Diversity Benefits:

- Avoid long-term commitment to a stack
- Enable the best technology for the team
- Avoid stack version conflicts between features or libraries
- Employ evolutionary design

Developers are passionate about technology and often have very strong opinions for or against a technology. These debates can go on for days, months, and even longer. All this discussion occurs before we can even begin building an application, and then we have to wait on the framework team to complete development.

With microservices we can use the best tool for the job. We can select the same or different technology stack for each individual microservice we are building. Different languages, stacks, data stores, and platforms can be used with each service. One of the services could be implemented using Node.js, and another service can be written in Java or .NET. Components of an application can require different versions of a framework or stack. One component of an application might not be able to use the latest version of a framework because another component's incompatibility with the

latest version.

As we can see in Figure 1.3, services can be implemented using a number of different languages and different data stores. A catalog service can be implemented in Java and benefit from Elasticsearch features, while the payment processing service can use ASP.NET and a transactional database with strong consistency and reporting features like SQL Server.

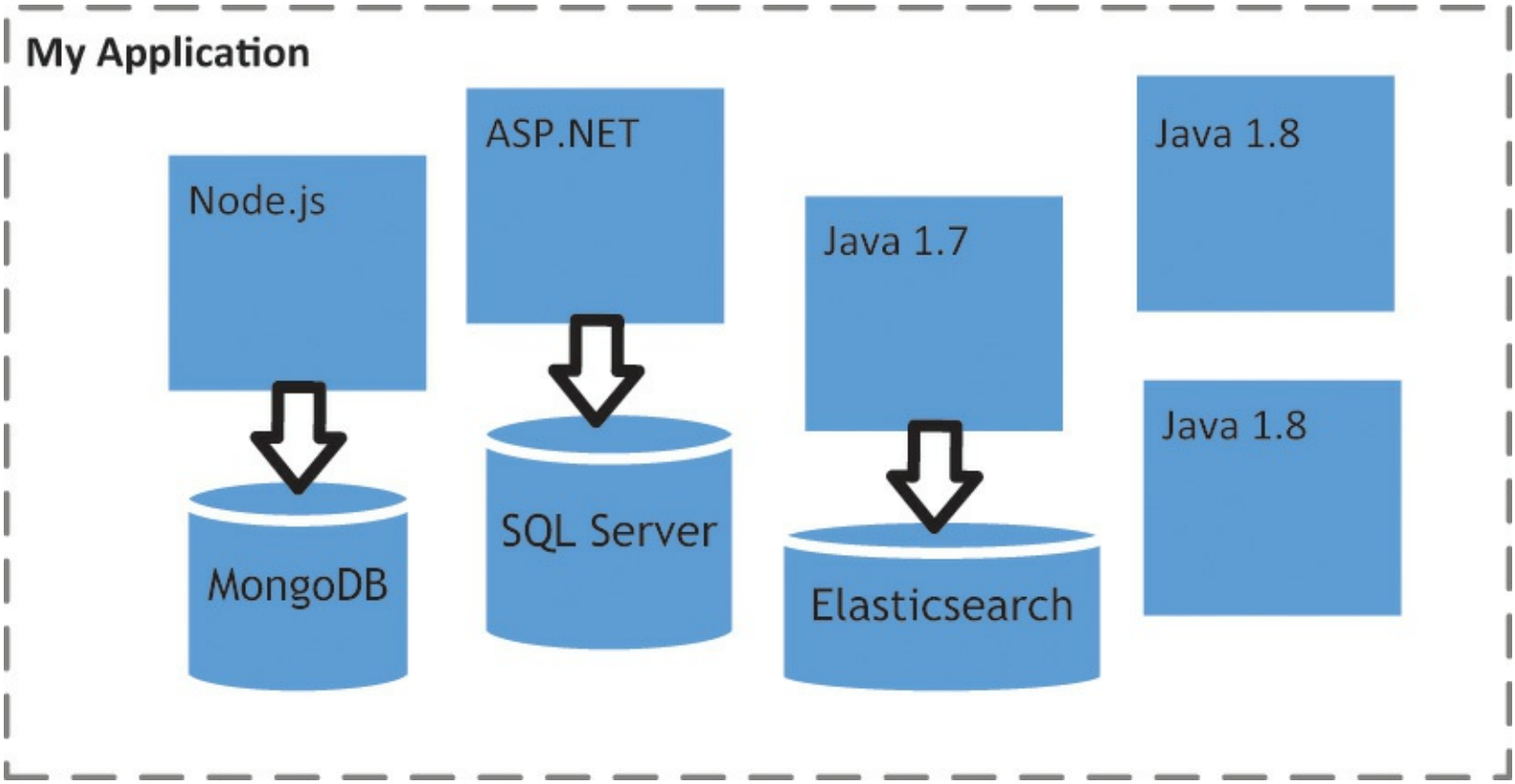

FIGURE 1.3: Services using different technologies and databases

> **Note**
>
> Just because we can use different languages and technologies across the various services does not mean we necessarily should.

The technology diversity and small size of the services enables evolutionary design. Technology is moving at an extremely rapid rate and leveraging new technologies in a monolith can be slow. If a decision is made to move to a new platform or technology, the entire application needs to be changed to support it. With the microservices architecture individual services can quickly adopt technologies that benefit the needs of that service without having to worry about the impact to other features in the application.

Small Focused Teams

Small teams are happy and productive teams. Structuring engineering teams at scale and keeping them productive can be quite challenging, although the recent DevOps movement has proven itself valuable to maintaining a productive team size for building and operating applications. Structuring engineering teams so that the developers are involved in the operations and are accountable for them has clear benefits in effectiveness. Applying this model to a very large application can be quite challenging, or nearly impossible. It's much easier to have developers involved and sharing the operational

responsibilities of the applications they are developing with smaller services.

Small individual services are ideally sized so that a team is able to take ownership of the complete service from design to operations. The isolation of the services maintains fairly clear lines or responsibility and ownership for the service. The responsibilities and concerns for the features of the service are easier to manage, as are the teams that are accountable for them. New developers joining a team can ramp up very quickly as they only need to understand a very small service.

Fault Isolation

In a monolithic application, a memory leak in one component, or even a third-party library, can affect the entire application and be hard to track down.

One of the benefits of a microservices architecture is the isolation of components or features. If one service crashes, it's quite possible the rest of the application can continue to operate until that service recovers. This of course assumes the service is implemented to handle the faults. Also, if one service is suffering from growing memory pressure due to a leak, it can be much easier to identify the service causing the issue. We might still be able to handle requests for the application even though one of its services is down. As depicted in Figure 1.4, even though one service is unhealthy and can be causing the process to crash, other services remain unaffected. As long as they are implemented correctly they can remain healthy.

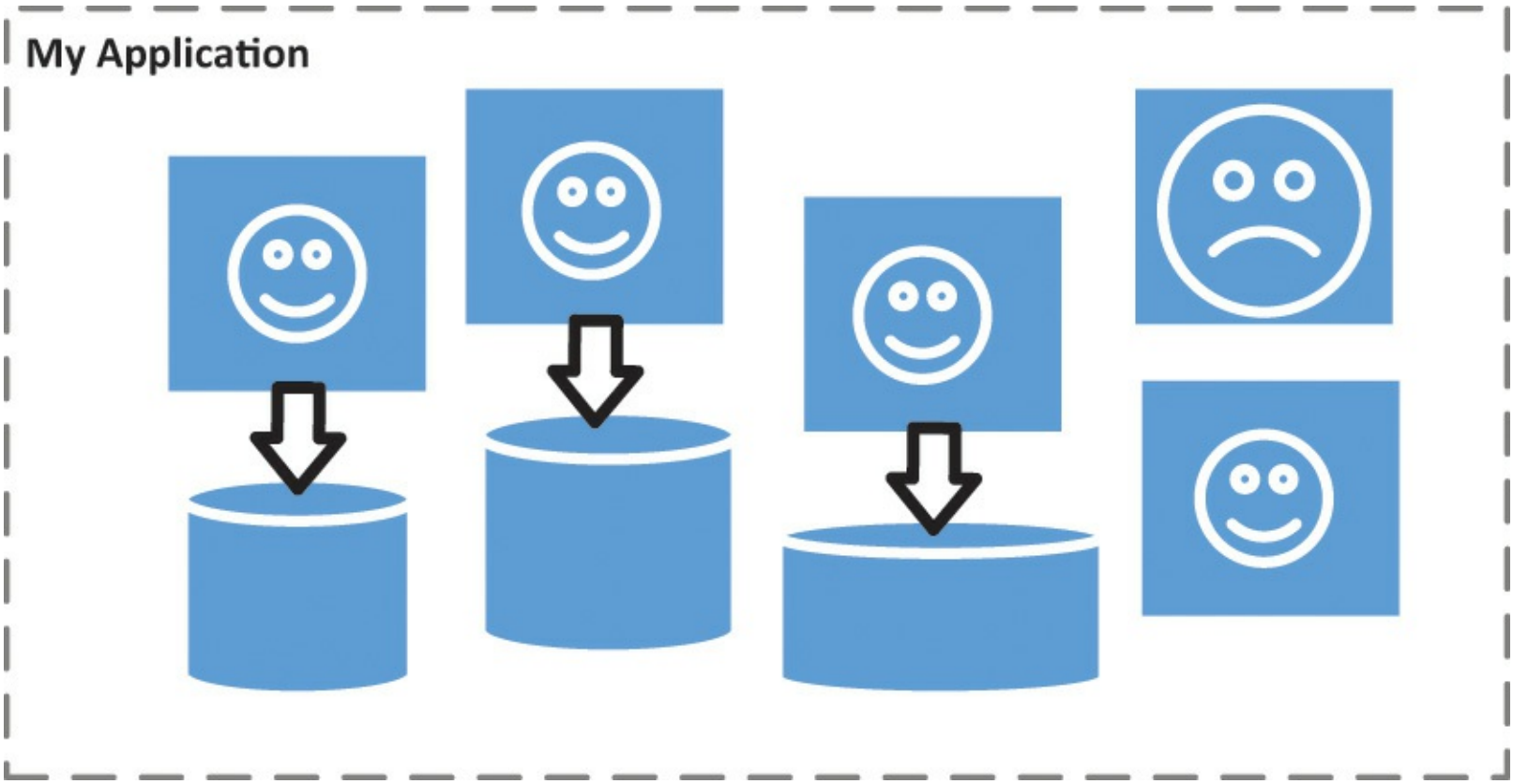

FIGURE 1.4: A failed service isolated from healthy ones

For example, if our recommendation service goes down, customers are still able to make their purchase. Customers might not be presented with recommendations and some potential sales could be lost, but not all of them. The authentication service going down can be more devastating, but instead of a complete service outage, users could still fill their carts, and when they are ready for check-out, can be presented with a screen warning them that if they are not be able to complete their purchase at that moment, they should try again later. Because we can quickly roll back when there is a failure, the

outage is short and the business impact, although potentially significant, is minimized.

Challenges

Despite the many benefits of microservices architectures, they come with their own set of challenges.

Although our complex application is decomposed into smaller, simple services, the architecture needed to run and manage them can present some big challenges. All these services need to work together to deliver the total functionality of our application.

Complexity

Distributed systems are inherently complex. Although the individual services themselves remain small and generally simple, the communication between the services and the management of the services becomes much more complex.

Keeping a monolithic application running on a small cluster can be a full time job in itself. In a microservices architecture, instead of one deployment, we are now deploying tens or hundreds of services, all of which need to be built, tested, and deployed, and that need to work seamlessly together. We need to ensure that all the services are up and running, have sufficient disk space and other resources, and remain performant. These services can even be implemented in multiple languages. All the services generally require load balancing and communication over synchronous and asynchronous messaging. A lot of the complexity in a microservices architecture is shifted into the infrastructure and may require experienced infrastructure developers.

Network Congestion and Latency

In our monolithic application, when a request comes in, it's processed by some instance of the application and a response is sent. Libraries are deployed with the application and calls to these libraries are fast and in-process. The application can make a small number of requests to a database or other services.

In the case of a microservices architecture, a single request to the application can result in multiple requests to a number of services. Each of the services could be making requests on their databases, caches, and other services. This I/O explosion can have a negative impact on the performance of an application and as a result optimizations in network communication becomes important. As shown in Figure 1.5, a call to a monolithic application on the left lands on a service instance and makes one or more calls to a database to handle the response. However, in the microservices architecture on the right side of the diagram, calls to the various components are now made across the network.

FIGURE 1.5: An external request results in many more internal requests when compared to a monolith

Most of the overhead in internal service calls are often in the data serialization and deserialization. In addition to data caching and replication to reduce the number of requests, efficient serialization formats can be used. Sometimes the same data is passed from one service to the next where it's deserialized and serialized multiple times. Using a common serialization format across the services can reduce some of these steps by enabling one service to pass data along to another service without having to reserialize all of it again.

Data Consistency

In our monolithic application, all data can be stored in a single database, and even if we partitioned it, we probably partitioned the data in such a way that we could still use the database to enforce referential integrity and use transactions. In our microservices application, the data is probably distributed across multiple databases, and in addition, they may be entirely different types of databases.

As we decentralize our data and move to a model where each service has its own schema or database, this causes data consistency and integrity challenges that can be hard to solve. Data in one service can reference data in another service where maintaining integrity between both is important, and if data changes in one database, it must change in the other.

What if we reference some data in another service that is deleted, or maybe the data changes and we need to be made aware of this? If we replicate data, how do we keep it consistent and who owns it? If we cache it, how do we invalidate caches?

Dealing with these consistency challenges, and concepts like eventual consistency, can be extremely hard to get right. Although dealing with these data consistency and integrity concerns can be quite challenging, there are some techniques we can use. We can use a notification service to publish changes to data where consumers can subscribe to these changes and updates. Another approach would be to set proper TTLs where some defined and acceptable inconsistency can be tolerated. We can implement the services in a way that they can deal with eventually consistent or inconsistent data.

Time to Live (TTL)

Time to live (TTL) is used to define the lifetime of some data in a cache or data store.

After this time has expired the data is considered stale and is then evicted/deleted. We can use this as a way to ensure data in a cache is refreshed after some period of time.

Many of these challenges are not new to microservices, as we commonly need to partition and replicate data in large systems as a way to increase scale and availability.

Testing

Service dependencies can complicate testing and generally require a more formal effort describing the service communication interfaces.

It can be difficult to recreate environments in a consistent way when you have multiple services evolving at a different rate. Whatever we deploy in to a staging environment will not exactly reflect what's in production. Setting up that many instances for the staging environment may not be worth the return. We can perform integration tests using mocks, service stubs, and consumer-driven contracts. A tool called Pact can be useful (https://github.com/realestate-com-au/pact).

A release should not end with the deployment of the service update. Testing in production is generally the way to go, and by placing more emphasis on monitoring we can detect anomalies and issues very quickly and roll back. Development teams should invest in automation and use strategies like blue-green deployments, canaries, dark canaries, A/B testing, and feature flags. This requires advanced DevOps; automation, and monitoring.

Integration and Versioning

When you begin to connect all the various services together, things might not necessarily work right away. The interfaces the services depend on are going to change in behavior and the contract. In a monolithic application it can be easy to refactor, change interfaces, and deploy the entire application together. The code consuming an interface can be deployed with the implementation of the interface. Changes to the interface could break the build and breaking changes in the behavior would show up when running integration or end-to-end tests. It's possible that when deploying a service, the other services that have dependencies on the one we are deploying could be break as a result.

Service versioning techniques can be used to ensure changes to a service do not break functions necessary to consumers. A service can use version information in the header or query string to ensure changes to a service do not affect existing consumers. It's also quite common to run multiple versions of a service side by side as separate deployments. Good automation and monitoring also enable us to immediately detect any breaking changes and quickly roll them back.

Service Discovery and Routing

In a monolithic application there is very little need to deal with discovery and routing of requests among business functions—libraries are compiled into a single application. We may have a dependency on a service or two, and those endpoints are commonly managed in the configuration. In our microservices environment, we need to be able to determine the endpoints for our dependencies and the route between them.

Although service discovery and routing can add some complexity, there are some very good solutions for dealing with this. In Chapter 5, "Service Orchestration and Connectivity," we will cover some of these solutions.

Monitoring and Logging

Many organizations struggle with monitoring a single monolithic application. Now let's multiply that by some number of services. We need to monitor what's going on inside all these different applications distributed across multiple instances, as well as all the interactions across the applications. Deploying basic monitoring and logging for a monolithic application can be fairly easy when compared to a distributed system.

In Chapter 7, "Monitoring," we cover monitoring and logging of microservices in more detail, including solutions to addressing this challenge.

Skillset and Experience

There are very few architects with microservices experience today, and these projects generally require developers with a broad set of skills. Having teams focused on a single problem domain can simplify the need for understanding the business domain, but an effective team still needs skilled multi-disciplinary team members. Also, as we have mentioned previously, a DevOps culture is valuable here, and finding experienced DevOps professionals can be challenging. Staffing a team with the unique skillsets and experience to be successful with a microservices approach can be a bit of a challenge today.

Although experienced engineers can be hard to find, given the excitement in the industry around microservices architectures, projects based on this approach can attract top talent.

Uptime Service Level Agreement

Earlier we talked about fault isolation and the benefits of a microservices architecture in avoiding a situation where one component of an application would take the entire application down. We mentioned the need to isolate our services from dependent services to avoid and contain any kind of faults in those services. Because the application itself is composed of a number of services, one of the things to consider is the combined SLA (Service Level Agreement) of those services.

> **Service Level Agreement (SLA)**
>
> An SLA is part of a contract that defines expected levels of service offered by a service or application. More often than not it includes a measure referred to as either *availability* or *uptime*. This is expressed as a percentage and often referred to as *three nines (99.9%), four nines (99.99%)*, or *five nines (99.999%)*. In the example of *three nines*, the SLA would stipulate that the application be up and available for 99.9% of a period time. If that period of time was 30 days, the application would be permitted to be down and unavailable for only 43 minutes and 12 seconds. If the SLA for the same application and period of time was *four nines (99.99%)*, the permissible downtime would be shortened to 4 minutes and 9 seconds.

If each of our services has an SLA of 99.9%, which means an individual service can be down for roughly forty-three minutes every month, we need to consider the fact that they could and are statistically likely to go down at different times. Now let's say the application is dependent on three services, each of which can be down for forty-three minutes a month at different times and still meet their SLA. This means the combined effective SLA for our application is now approximately two

hours of outage each month.

Detecting the failure of a dependent service and gracefully dealing with it can help keep the application's overall SLA higher. The application might be able to temporarily provide service with limited functionality in the event that one of its service dependencies is experiencing issues. These are some important considerations when designing the application.

Best Practices

Several best practices for the design and implementation of microservices architectures have been established by organizations working with microservices over the years. Before diving into the design and development of our microservices, we will cover some of the best practices and some of the challenges in building microservices.

Encapsulation

Encapsulating the internals of a service through a carefully defined service boundary that does not expose internals of a service is important. Maintaining loose coupling in microservices is important to realizing the benefits of a microservices approach. If we are not careful, our microservices can become coupled. Once these services become coupled we can lose a lot of the value in a microservices architecture, like independent lifecycles. Deploying and managing a microservices-based application with tightly coupled services can be more complex than deploying and managing a monolith. If the deployment of a service needs to be closely coordinated with other services, there is likely coupling as a result.

Common causes of service coupling:

- Shared Database Schema
- Message Bus or API Gateway
- API design
- Rigid Protocols
- Exposing service internals in the API
- Organizational coupling (Conway's Law)

> **Conway's Law**
>
> Conway's Law generally states that the design of a system reflects the communication structure of the organization.

Shared databases and schemas are common causes of coupling. We start out building our microservices, and because our microservices all use a SQL database, and there is some overlap in the schema, we simply have a bunch of our services share a database. All of our data is related in some way, so why not? It would appear that we can deploy and manage our services independently until we need to change something in the database, and then we need to coordinate a deployment with the other services and their teams. It might sometimes happen that a database schema never changes; however, more often than not they do need changing over time. There can be good reasons for two or more microservices to share database schema and still remain separate microservices, but it's important to understand what the trade-offs are.

Another common cause of coupling that was prominent in past implementations of service-oriented architecture was an Enterprise Service Bus (ESB). The service bus often contained schemas, transforms, and business logic that were tightly coupled with the services that integrated with them.

Not only do we need to ensure the design and implementation enables us to reduce or eliminate the friction between our services so we can move quickly, it's important that the teams themselves do not become coupled.

DevOps Principles and Culture

If you are considering microservices, you really should have a strong devops practice in place, or plan on putting one in place. A good devops culture is one that institutes freedom and responsibility, and reduces process. Devops often includes engineering and operations functions in the same team. Many of the benefits we discussed earlier are as much a result of strong devops practices as they are a result of microservices architecture:

- Collaboration
- Automation
- Monitoring

In a devops environment teams are cross-functional, consisting of developers, QA, DBAs, and operations. The shift to devops is often made to accelerate feature delivery and provide stability in that delivery. Context is not lost among teams as the application transitions from engineering to operations, or in feedback from operations to engineering. Engineering and operations people work together to build, design, deploy, and manage the application. The team is accountable for not only the development but also the business operations of the service. This accountability helps improve all those things about the application that make it easier to manage in production and more stable.

Benefits of devops:

- Faster delivery of features through continuous software delivery
- Less complex problems
- Faster resolution to problems
- More time to add features

As mentioned, many of the benefits of devops have also been presented as benefits of a microservices architecture. The two are complimentary and the microservices architecture enables devops at scale. Devops success is also dependent on building a devops culture throughout the organization, and this can be a significant organizational shift.

Automation

Use of automation is recommended when building a monolithic application, but it is absolutely necessary for microservices applications. Given the increased number of services that need to be built and deployed, it's even more important to automate the build and deployment of the services. A widely accepted principle is that the first thing a team needs to build when starting a microservices project is the continuous delivery pipeline. Manual deployment takes a lot of time, and is the primary cause of failures in production. Multiply this issue by the number of services, and the chances of human error go up significantly. It is much better to write code or build a deployment pipeline than be responsible for manual deployments.

The automated pipeline can make these functions smoother and less error-prone:

- Builds
- Tests
- Deployments
- Rollback
- Routing & Registry
- Environments

All things should be automated and there should be no need for a deployment manual. Not only should we be concerned with automating the delivery of the application, we should consider automation for development. It should be easy for developers to provision environments, and friction due to a lack of automation in this area can impede development.

Monitoring

Good monitoring is important to keeping any application healthy and available. In a microservices architecture good monitoring is even more important. We not only need to be able to monitor what's going on inside a service, but also the interactions between the services and the operations that span them.

- **Use activity or correlation IDs:** These are used to associate events to a specific activity or transaction.
- **Use a common log format:** This enables better correlation of events across services and instances.
- **Collect and analyze logs:** By collecting logs from the various services and instances, we can analyze and query across multiple services and instances of a service.
- **Consider using a sidecar:** A sidecar is simply an out-of-process agent that is responsible for collecting metrics and logs. We can run this in a linked container.

In addition to these points, all the standard monitoring tools and techniques should be used where appropriate. This includes endpoint monitoring and synthetic user monitoring.

See Chapter 7 for more information on best practices for monitoring and logging with microservices architectures.

Fault Tolerance

Just because we move to a microservices architecture and have isolated one component from another in either a process, container, or instance, it does not necessarily mean we get fault tolerance for free. Faults can sometimes be more prevalent and more of a challenge in a microservices architecture. We split our application into services and put a network in between them, which can cause failures. Although services are running in their own processes or containers and other services should not directly affect them, one bad microservice can bring the entire application down. For instance, a service could take too long to respond, exhausting all the threads in the calling service. This can cause cascading failures, but there are some techniques we can employ to help address this.

- **Timeouts:** Properly configured timeouts can reduce the chance of resources being tied up waiting on service calls that fail slowly.

- **Circuit breaker:** If we can know a request to a service is going to result in a failure then we should not call it.
- **Bulkheads:** Isolate resources used to call-dependent services.
- **Retry:** In a cloud environment, some failed service operations are transient and we can simply detect these and retry the operation.

Let's have a look at these patterns and what they are.

If an application or update is going to fail, it's better to fail fast. We cannot allow a service we are calling to cause a failure that cascades through the system. We can often avoid cascading failures by configuring proper timeouts and circuit breakers. Sometimes, not receiving a response from a service can be worse than receiving an exception. Resources are then tied up waiting for a response, and as more requests are piled up, we start to run out resources. We can't wait forever for a response that we may never receive. It would be far better if we never send requests to a service that is unavailable. In Figure 1.6, Service E is failing to respond in a timely manner, which consumes all the resources of the upstream services waiting for a response, causing a cascading failure across the system.

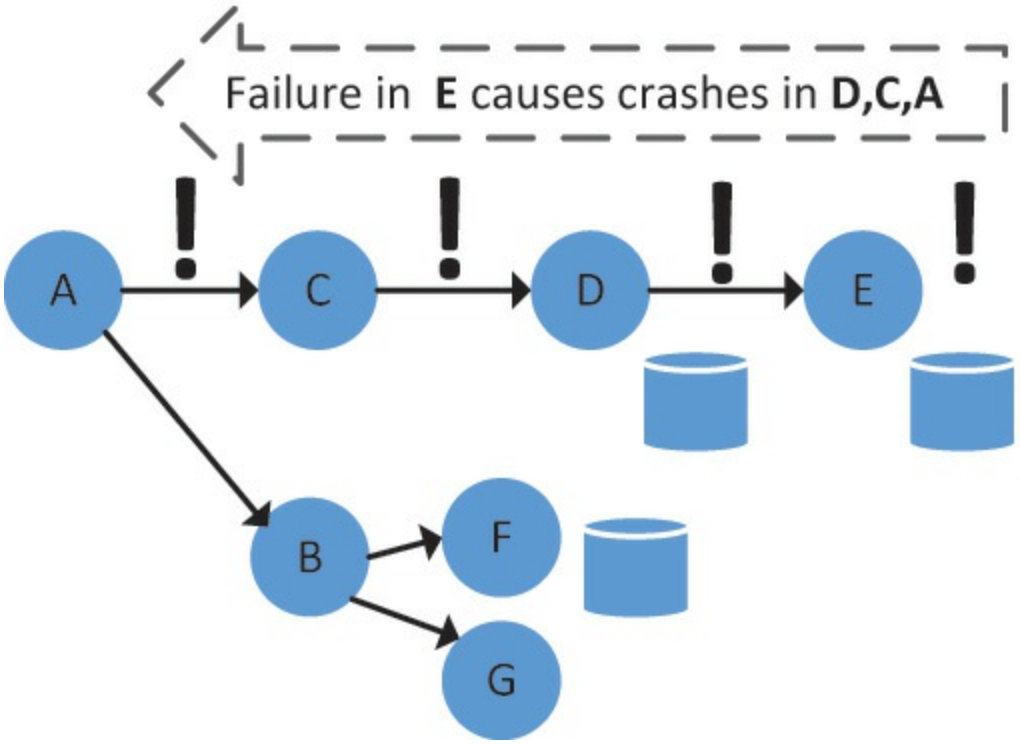

FIGURE 1.6: Cascading failures

Quite often a service will have a dependency on multiple services, and if one is behaving badly we don't want that to affect requests to the others. We can isolate the service dependencies and place them in different thread pools, so if Service A starts to time out, it does not affect the thread pool for Service B. This protection is similar to that of bulkheads in a ship; if we get a leak in the hull of the ship we can seal off that section to avoid sinking the entire ship. As we see in Figure 1.7, calls to different services are isolated into bulkheads. This means that if service 'H' is behaving badly, it's not going to affect calls to service 'G' or other resources in the service.

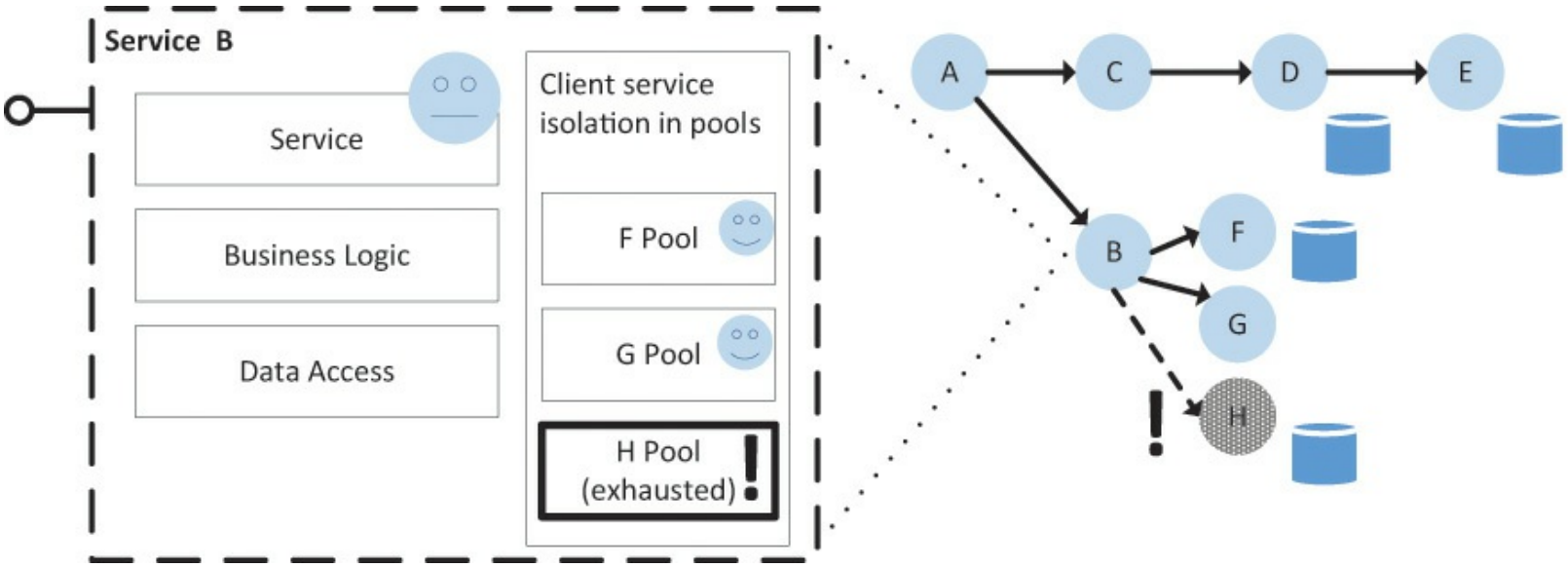

FIGURE 1.7: Service dependencies isolated into bulkheads

Some exceptions from service calls are transient in nature and we can simply retry the operations. This can avoid exceptions and the need to handle the entire request again. This typically involves a detection strategy to determine what exceptions are transient, and then to define a strategy for how to handle it, such as how many times to retry the operation and how long to wait before retrying. We do need to be careful that our retry logic does not overwhelm a failing service, however.

More information on retry patterns is available in "Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications," an article available on MSDN (https://msdn.microsoft.com/en-us/library/dn589788.aspx). Additional guidance implementing this pattern can be found in the Microsoft Patterns & Practices repository (https://azure.microsoft.com/en-us/documentation/articles/best-practices-retry-general/).

If we can know that a request to a service is going to fail, it's better to fail fast and not call the service at all. Resources are required to make a service call, and if the service is unhealthy and is going to fail, then there is little point in calling it. Also, the service might be trying to recover or scale up to meet some increased demand, and retrying the call is only making it worse. To address this, we can utilize a pattern called the circuit breaker pattern. If the calls exceed some threshold, we move to an open state, or trip the breaker just like the circuit breaker in your home. When the circuit breaker is open, we stop sending requests until it's closed. Like the wiring and devices in the home, a circuit breaker can help protect the rest of the system from faults. Circuit breakers are not triggered automatically exclusively when a service is failing, but can sometimes be triggered manually when a service needs to be taken offline for maintenance or to recover.

More information on the circuit breaker design pattern is available in "Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications" (https://msdn.microsoft.com/en-us/library/dn589784.aspx).

Summary

There are a lot of advantages to a microservices architecture. The autonomy of services that it provides can be worth the cost of segregating the services.

- Each service is simple.
- Services have independent lifecycles.

- Services have independent scaling and performance.
- It is easier to optimally tune the performance of each service.

A microservices architecture does come at a cost and there are a number of challenges and trade-offs that must be considered.

- More services must be deployed and managed.
- Many small repositories must be administered.
- A stronger need exists for sophisticated tooling and dependency management.
- Network latencies are a risk.

Organizations need to consider whether they are ready to adopt this approach and what they need to change to be successful with it.

- Can the organization adopt a devops culture of freedom and accountability?
- Does the team have the skills and experience?
- Is continuous delivery possible?
- Is the organization utilizing cloud technology (private or public) so it can easily provision resources?

In the following chapters we will discuss containers and how they can be used in a microservices architecture. We will also work through designing microservices and all the important processes for developers, operations, and releasing an application based on a microservices architectural style.

2. Containers on Azure Basics

In this chapter, we will lay down the foundation of basic container knowledge that we will need throughout the book. This chapter will start with a comparison of VMs, containers, and processes, and will talk about containers on Azure. We will create an Azure VM with containers on it and use it as our training environment to make ourselves familiar with basic container operations.

In the subsequent chapters, we will expand our container knowledge based on real world examples.

VMs, Containers, and Processes

By now, almost all public cloud vendors offer a container strategy, so what is it that makes containers so popular?

To see the value of containers, we should look at some of the most common challenges with application development and application life cycle. Customers have repeatedly asked for guidance on how to handle environmental differences, how to make environments portable, how to increase the density of running services on one machine, and how to isolate services. Containers are the right tool to solve these issues.

Containers enable us to slice up an operating system so that we can securely run multiple applications on a single operating system. If we want to run five applications or services on an operating system we would create five containers. Each container has its own isolated view of things like network stack, filesystem, process tree, and so on. For example, every container on an operating system gets its own root directory (/), its own eth0 network interface, its own PID 0. Each container is unaware that there are other containers sharing the same operating system. To achieve this, Linux leverages a couple of kernel features: namespaces and control groups. Namespaces enable the different components of the operating system to be sliced up and thus create isolated workspaces. Control groups then enable fine-grained control of resource utilization, effectively preventing one container from hogging all system resources. They achieve this by partitioning CPU time, system memory, disk bandwidth, and network bandwidth into groups and assigning tasks to it. We also might want to monitor resource consumption for any application assigned to a control group. Chapter 7 covers monitoring in more detail.

As a result, we can think of containers as encapsulated, individually deployable components running as isolated instances on the same kernel that is leveraging operating system level virtualization. This means each applications, its runtime, dependencies, system libraries, and so on, running inside a container has full, private access to its own isolated view of operating system constructs.

From a developer point of view, we can just package our application and dependencies into a container and deploy it to any environment that supports containers. By doing this we also make our application easy to update and upgrade, as well as easily portable from one environment to another. For example, a container can make an application easily portable from a development environment on the desktop, to a test environment in the cloud, and then to an on-premises production environment.

There are multiple container technologies available, but over the last two years Docker has emerged as the de facto standard. For the remainder of this book, we will focus on Docker containers.

When Would We Use a Container Over a Virtual Machine or a Process?

This is a valid question, especially considering the fact many companies have heavily invested in virtualized environments on premises and in the cloud. Hypervisor virtualization technologies have proven to be secure and compatible, and we could even look at a virtual machine as an encapsulated, individually deployable component.

The answer to this question really lies in the scenario. Not every scenario should, or even can be, solved with containers, so it is important to understand where containers are a good choice over virtual machines.

Let's have a closer look at a scenario that enables us to compare virtual machines and containers.

Consider a cloud service that needs to calculate the average rating for a given picture. In large social networks or picture-sharing applications, the average rating does not need to be updated immediately after a user has rated the picture. It is sufficient if the average rating is updated once or twice a day.

A good technical and economical way of implementing periodic tasks is to run a background job on an existing compute cluster. This background job only needs to run once or twice a day, for a short period of time, and then immediately terminate.

If we used a virtual machine for this job, we would not want to have it up and running all the time as this incurs unnecessary costs. One way of implementing this scenario is to start a virtual machine instance on demand when the job needs to run. Although that sounds good, there are a few issues to consider.

First, the virtual machine can take a considerable amount of time to start—there is a full operating system to boot. There are times when booting the VM takes longer than the actual runtime of the job.

Second, the size of the virtual machine can be an issue. A virtual machine contains an entire operating system, which can easily be several gigabytes in size. For example, the Windows Server-based Visual Studio virtual machine images in the Azure gallery are all more than 128GB in size. Copying this image across a network, if they are kept in a central image repository for example, will take a lot of time.

Third, scaling of virtual machines has its challenges. Scaling up virtual machines requires a new, larger virtual machine (more CPU, memory, storage, and so on) to be provisioned and booted. Scaling out also runs into the startup time problems of virtual machines. In our example, the second virtual machine can take longer to provision and boot than to run the actual job, resulting in it coming online too late.

Another consideration is the economics of virtual machines. Virtual machines use considerable amounts of system resources such as memory, CPU, and disk space. This limits the number of virtual machines that can run on a single host machine; running lots of virtual machines on a single host requires the use of expensive hosts with lots of CPU and memory.

Scaling Up vs. Scaling Out

Scaling up, also referred to as vertical scaling, means adding more resources to a machine. In most cases, this refers to adding more RAM. This is the less-preferred option in cloud environments as it is usually more expensive, and the demand could outpace the capacity before the new machine has been provisioned. Scaling out, also

referred to as horizontal scaling, means to add more instances of the machines to an environment. This is the preferred scaling mechanism in cloud environments.

As we have seen, virtual machines are not ideal for scenarios where new instances need to be spun up quickly.

The next idea that comes to mind is why not just build a process on one of the virtual machines in the existing computer cluster? (A cluster is a group of VMs.) Processes start fast, dynamically allocate resources such as RAM, and are efficient at sharing these, and so it looks like a good fit. The downside with processes is that they are not well-isolated from the rest of the environment, and can quickly result in noisy neighbor situations, potentially compromising the entire virtual machine if the code was not written well.

Noisy Neighbor

The noisy neighbor problem describes a situation where one application takes up or blocks much of the resources that are shared amongst other applications on the same infrastructure. This negatively affects the execution of other processes and applications.

Based on what we know about containers, they seem to be the ideal choice for this scenario. We can spin up a background-processing task quickly, run it isolated, and scale it out quickly.

We can also see the difference between running the rating service in a VM and running it in a container by looking at the figures following. Figure 2.1 shows the high-level topology for a machine (Host OS) hosting two virtual machines (Guest OS) with applications running on virtual machines. The box "Rating Service" represents our rating service on a virtual machine. As we can see, it shares resources with application App on the same virtual machine. If there is a problem with the background task, it can jeopardize the entire machine.

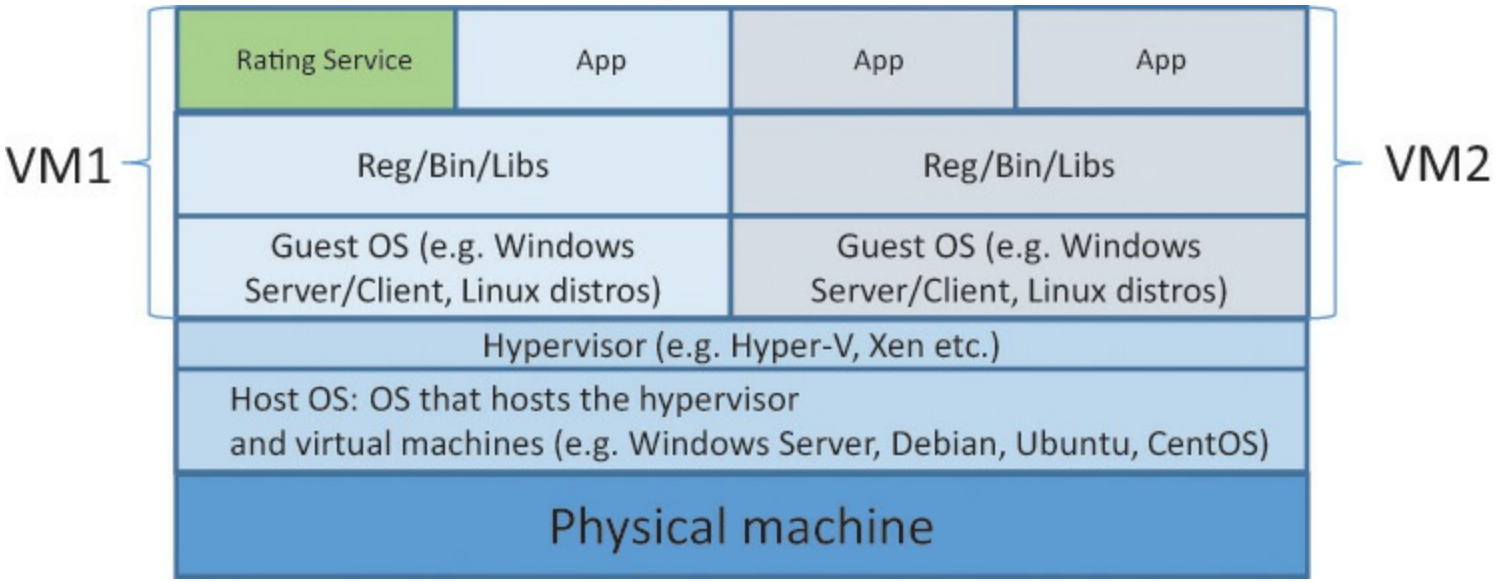

FIGURE 2.1: Rating service running in a virtual machine on a host with two virtual machines

Figure 2.2 shows the rating service running in a container. The container encapsulates everything that is needed for the background task including all libraries, dependencies, and files. It runs as an isolated user process on the same Host OS, sharing the same kernel as other containers. If there is a problem with the rating service it would have no impact on the other applications running in

containers as they are isolated from each other.

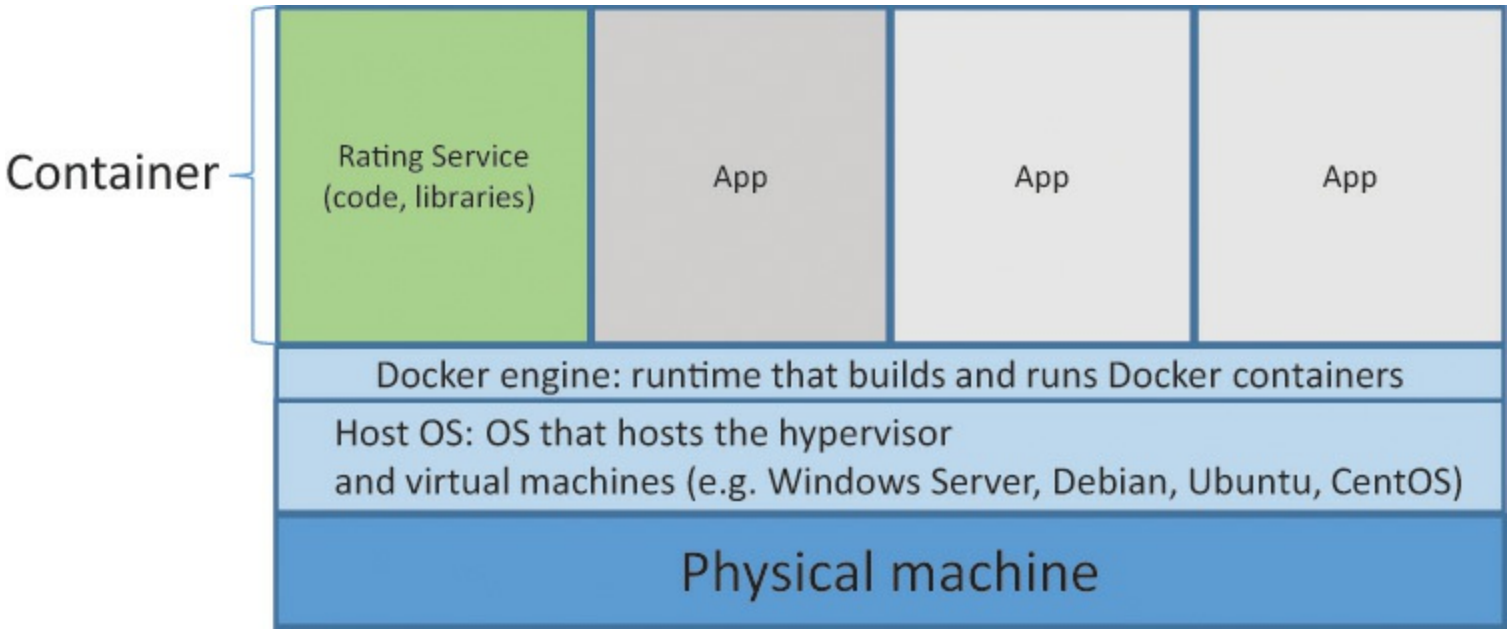

FIGURE 2.2: Rating service running in a container on a host with multiple containers

We used the example of the background-processing task to illustrate how to think about virtual machines vs. containers and processes, but there are other scenarios where containers can be a good fit.

Containers are a great choice in scenarios that require reactive security. As containers are isolated units, the attack space is limited, and due to the speedy boot and shutdown of containers, we can kill them and replace them quickly.

> **Reactive Security**
>
> Reactive security refers to actions that are taken **after** discovering that some systems have been compromised by a malicious program or script.

As mentioned previously, by using containers we make our applications portable. The very same application container can run on a laptop, a physical or hypervisor server, or in a cloud environment. In particular, dev-test and DevOps benefit hugely from this kind of portability. Chapter 6 covers DevOps practices in detail.

Besides dev-test, high density and microservices scenarios benefit the most from containers, and this is what we will learn in this book.

Containers on Azure

Within Microsoft Azure, virtual machines offer many platforms to choose from. These include many Linux distributions such as CoreOS, SUSE, Ubuntu and Red Hat, as well as Windows-based systems. As mentioned before, we will focus on Docker containers in this book. Docker is currently the most popular container technology and the Docker ecosystem is growing at a very fast pace. Microsoft has entered into a partnership with Docker, Inc., which contains an agreement that the open-source Docker engine that builds and runs Docker containers is available in Windows Server 2016. Windows Server containers use the Docker runtime and enable developers who are familiar with the Windows ecosystem to develop container-based solutions. At the time of writing, Windows

containers are still in preview. We will have an online chapter about Windows containers after they debut in Windows Server 2016.

> **Can I Run Linux Containers on Windows and Vice Versa?**
>
> The short answer is no. Remember that containers are based on operating system level virtualization, and as such share the kernel of the host they are running on. Windows and Linux systems have fundamentally different kernels, and as a result you can only run Windows containers on Windows systems and Linux containers on Linux systems.

Creating an Azure VM with Docker

The Docker web site (https://www.docker.com/) provides a very good overview of the Docker architecture and how it works, in case you want to gain a deeper understanding of Docker internals. For the purpose of this book, we do not need to understand all the details of Docker, so let's have a look at the parts that are relevant for us.

If we want to run Docker containers, we need a Docker host machine. The piece that is really needed on the host machine is the Docker daemon. This is responsible for managing Docker images and containers on the host. To communicate with the Docker daemon, we use the Docker client. The Docker client is really just a binary that talks to the Docker Remote API implemented in the Docker daemon. The Docker client can be used remotely, but can also be installed on the Docker host alongside the Docker daemon.

The fundamental features when working with Docker are images, containers, repositories and volumes. The best way to explore those features is to work with them. Over the next few pages, we will discuss how to set up a VM in Azure as a Docker host machine, and explore fundamental Docker commands.

There are several ways to install Docker on an Azure VM:

- **Command Line Interface**: The Azure CLI tools on Windows and Mac OS X enable us to create virtual machines that include Docker.
- The command "`azure vm docker create -location "<location>" [options] <dns-name> <image> <user-name> [password]`" creates an Azure VM and uses the Docker extension to install Docker. At the time of writing there is no command line option for an SSH public key; you have to use a username and password instead.
- Another CLI option is to use Docker Machine. Docker Machine is part of the Docker toolbox and simplifies the creation of Docker hosts on your computer, on cloud providers, and inside your own data center.
- The command

Click here to view code image

```
docker-machine_linux-amd64 create -d azure
      --azure-subscription-
id="<subscriptionID>"
      --azure-subscription-cert="mycert.pem"
  machine-name
```

creates a Docker host on Azure.

- **Azure Resource Manager (ARM) templates**: Azure Resource Manager enables you to declaratively describe your environment in a template, and deploy and manage it as a single unit. In Chapter 5, we will have a closer look at Azure Resource Manager. There are many templates for Docker. including quickstart templates available on GitHub (https://github.com/Azure/azure-quickstart-templates) that enable users to provision an environment directly into Azure.
- **PowerShell**: PowerShell can be used to deploy Azure Resource Manager templates.
- The command "`New-AzureResourceGroupDeployment -Name $Name -ResourceGroupName $RGName -TemplateUri $vmTemplateURI --TemplateParameterObject $parameters -Verbose`" will provision a new Azure environment based on the template that has been submitted.
- **Azure Portal**: Provides a UI for creating Azure VMs, including one with Docker installed.

There is no Azure VM image that has the Docker daemon preinstalled; in all cases Docker is installed as a VM extension to the Azure VM.

> **VM Extension**
>
> A virtual machine extension is a software component enabling in-guest management customization tasks such as adding software or enabling debugging and diagnostics scenarios. Extensions are authored by Microsoft and trusted third parties like Puppet Labs and Chef. The set of available extensions is growing all the time. The Docker extension installs the Docker daemon on supported Linux-based virtual machines. Extension can be installed in through the portal, command line tools (such as Azure CLI), PowerShell, or Visual Studio.

In this chapter we use the easiest way possible: we provision an Azure VM from an image that will install Docker as part of the virtual machine provisioning process through the Azure Portal.

> **Install Docker on a Development Machine**
>
> To install Docker locally on a machine (Mac OS, Linux, or Windows) we should use Docker Toolbox. Toolbox is a collection of Docker tools bundled in a single installer. For more information go to this URL: https://www.docker.com/toolbox. For a graphical user interface we can install Kitematic from Docker (for Windows and Mac). Chapter 4, "Setting Up Your Development Environment," covers setting up Docker locally in more detail.

Before we create the Azure VM we need to generate an SSH key that we will use to connect to the Azure VM.

Generating an SSH Public Key on Windows

The Windows operating system does not ship an SSH client or utilities to create cryptographic keys. In this book, we use the Git Bash third-party tool, but we could also use Git Shell or Putty. We can install Git, Git Bash and Git GUI from http://www.git-scm.com/download/win.

Let's start with creating the public key for our SSH connection. On Windows open Git Bash and enter the following command:

```
ssh-keygen -f dockerhostkey
```

Next we are asked to enter a passphrase. We don't have to enter one, but using a passphrase provides a little more security. It prevents anyone obtaining the key from using it without the passphrase.

The Git Bash console should now look similar to the one shown in Figure 2.3.

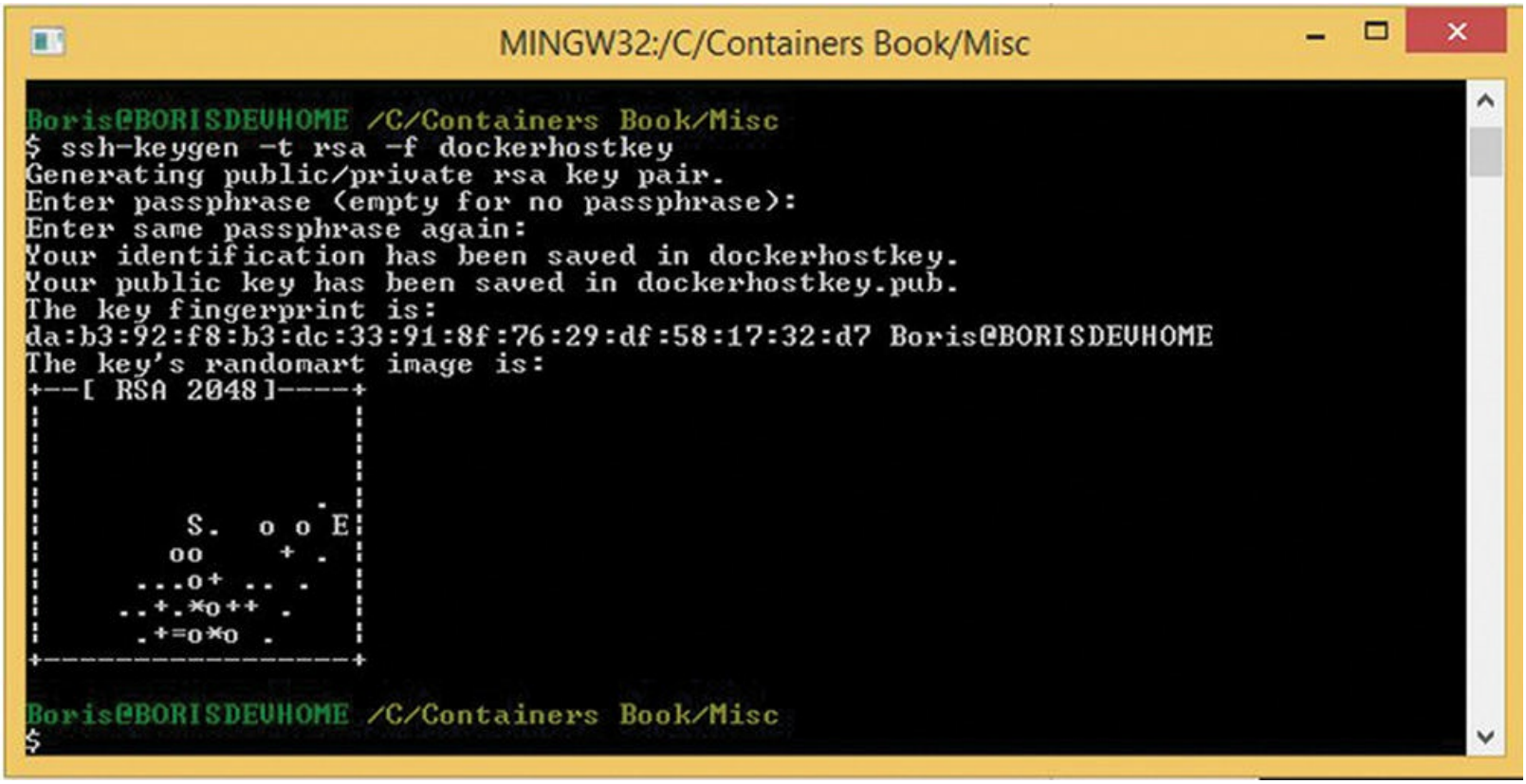

FIGURE 2.3: SSH key generation in Git Bash

The next step is to go to the directory from where you have executed the command. In this case, the directory to go to is c:\Container Book\Misc. The directory contains two files:

- **dockerhostkey:** This file contains the private key; never share this file with anyone.
- **dockerhostkey.pub:** This file contains the public key; you can share this file with others. We need to open this file with a text editor of our choice and copy the key.

When opening the *.pub file on Windows in Notepad we need to make sure to copy the key with a new line at the end (we can delete the username@machine first and then move our cursor to the last character of the key/file and add a new line). Without having a new line, the image will fail to provision, as Linux cannot generate a *.pem file due to a malformed key during provisioning.

Generating an SSH Public Key on Mac OS X

On Mac OS X, we do not need to install any additional tools as SSH comes with the OS. In the terminal, we navigate to the directory we want to use as our working directory and type:

```
ssh-keygen -f dockerhostkey
```

Our terminal window should now look similar to the one shown in Figure 2.4.

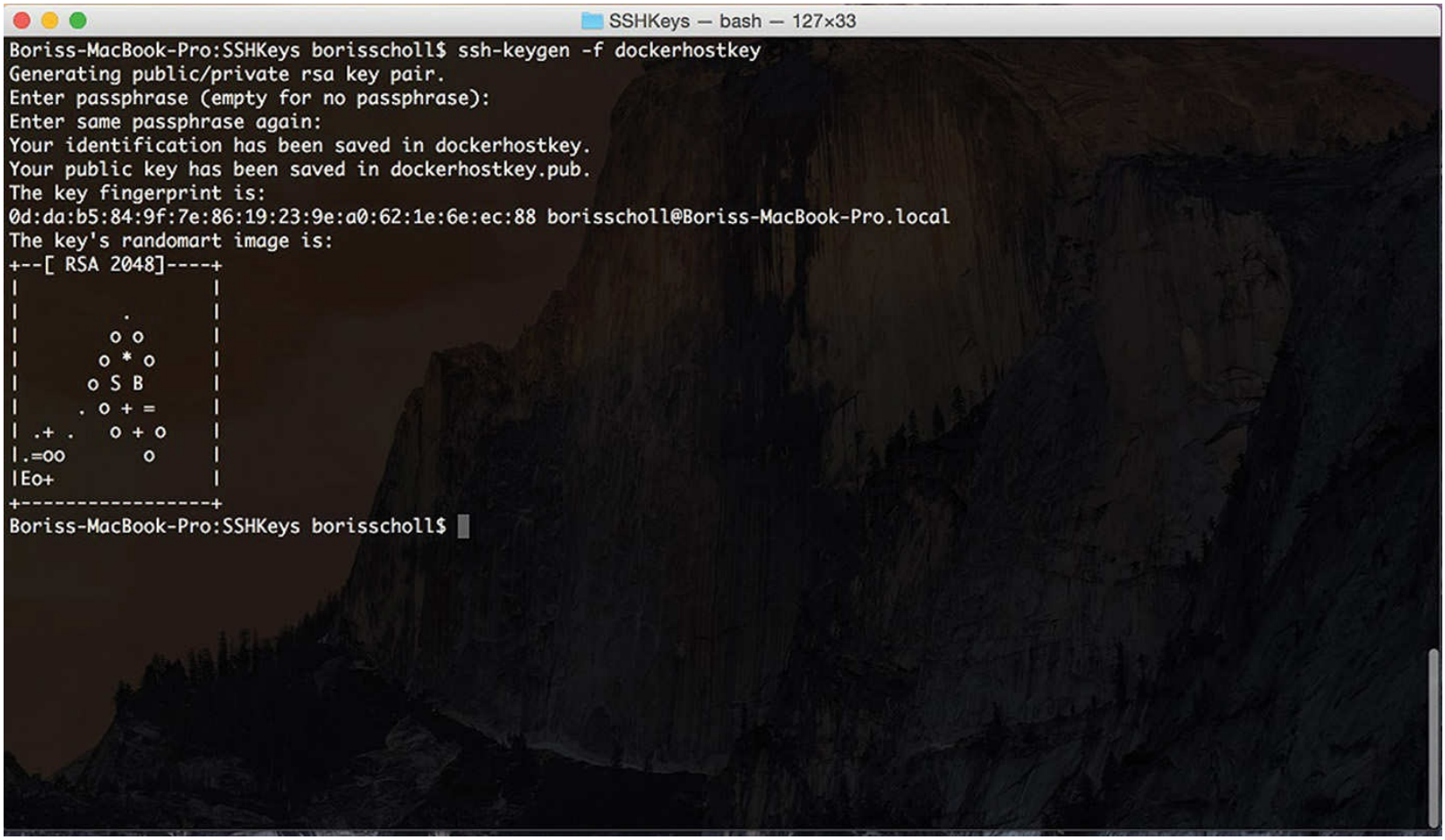

FIGURE 2.4: SSH key generation in MAC OS terminal

Choosing a Virtual Machine Image

Now that we have the SSH key we can provision the Azure VM from the Docker on the Ubuntu Server image using the Azure Portal. The portal can be accessed at http://portal.azure.com.

Sign in to the Azure Portal by entering http://portal.azure.com in your browser. Once the homepage has loaded, click the "+ New" button in the upper left corner, enter `Docker` in the "Search the marketplace" text box and hit "Enter." Figure 2.5 shows the portal displaying the search results for Docker.

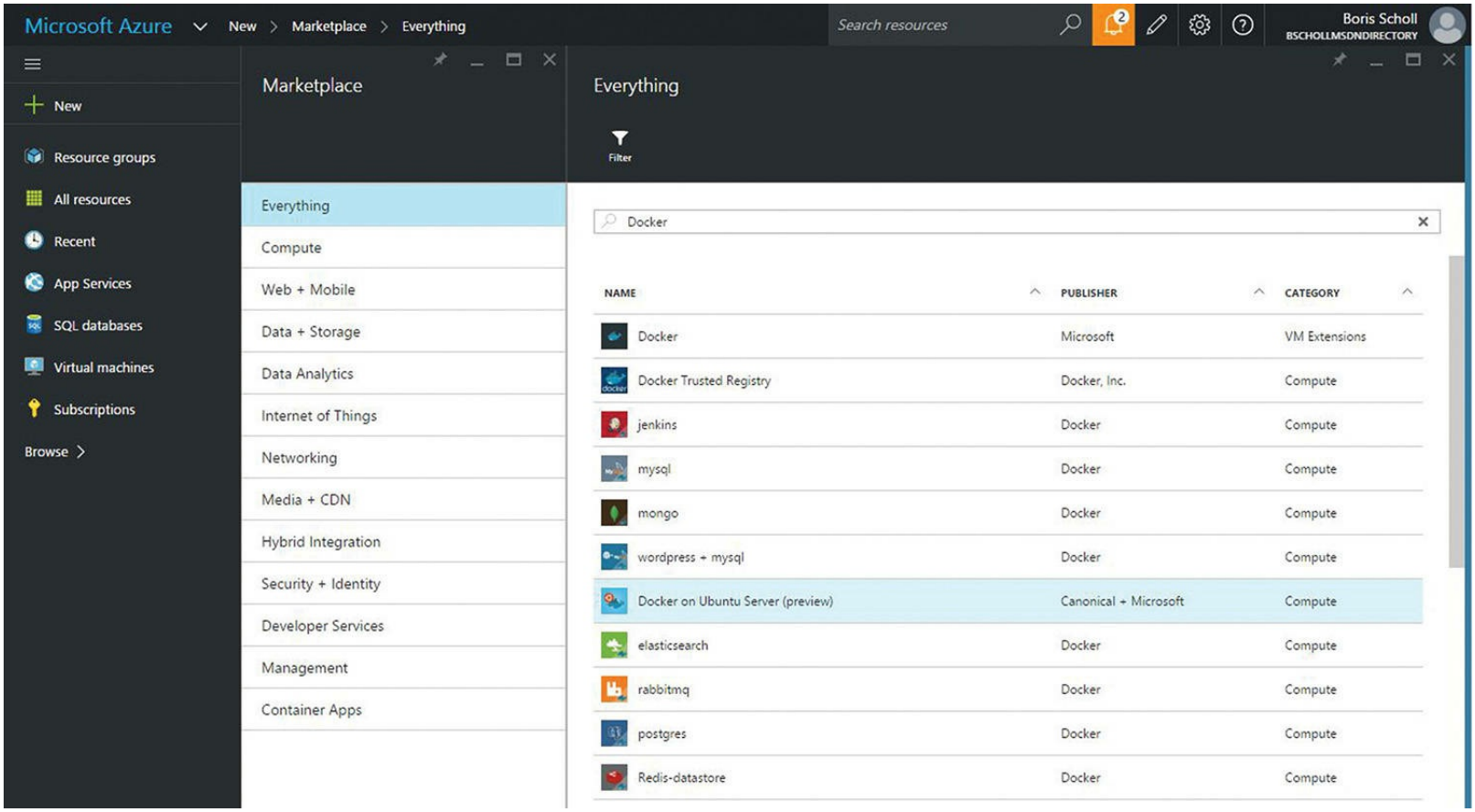

FIGURE 2.5: Search results for Docker in the portal

Next, we click the "Docker on Ubuntu Server (preview)" image. Canonical and Microsoft Open Tech created this image specifically to make it easy to set up an Azure VM as a Docker host. Clicking on this image will take us to a new page, which is called blade in the portal, displaying information about the Azure image. Figure 2.6 shows the Information blade.

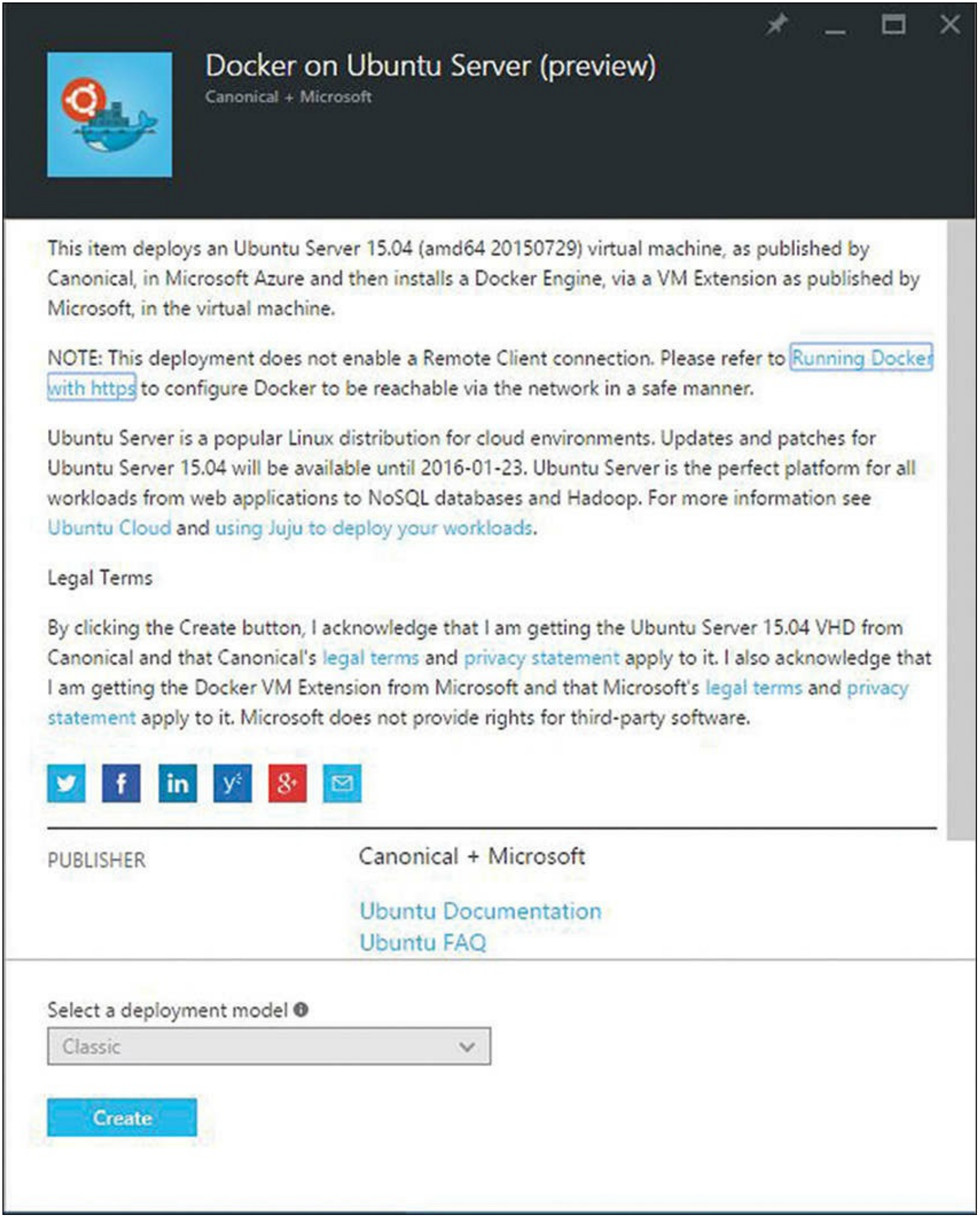

FIGURE 2.6: Information blade

Deployment Model

Azure knows two deployment models: classic and resource manager. This image is based on the classic deployment model, which means it does not use Azure Resource Manager. The preferred model is Azure Resource Manager, but at the time of writing, the image was for the classic model. If you want to create an Azure VM as Docker host using the resource manager model, you can use the template "docker-simple-on-ubuntu" under Azure-Quickstart-Templates on GitHub.

Clicking the "Create" button opens up the Create VM blade as shown in Figure 2.7.

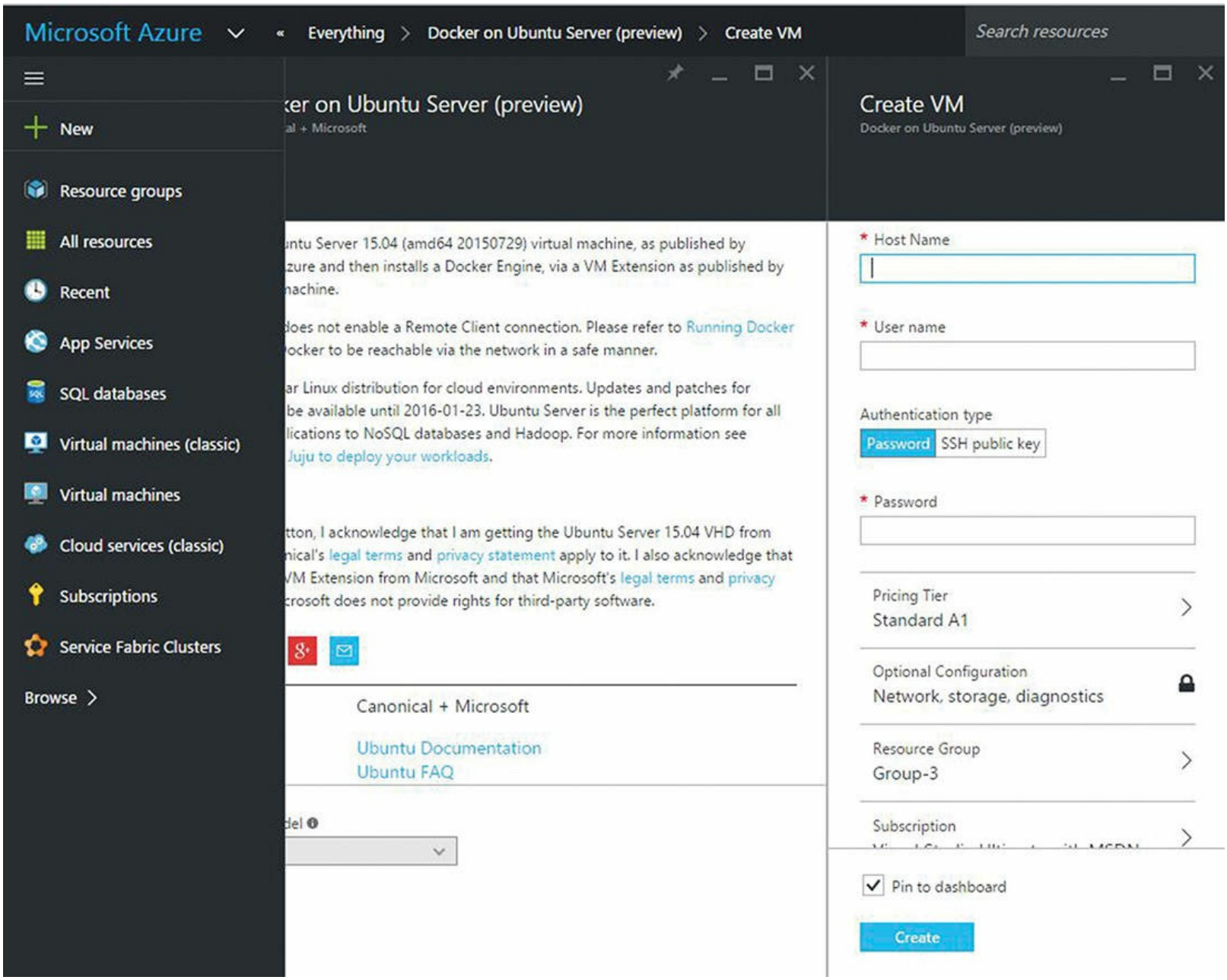

FIGURE 2.7: Create new VM blade

First we need to enter the Host Name for our virtual machine. For the purpose of this exercise, we can just call it dockerhost. The next textbox requires us to provide a User name.

We need the User name to connect to the Linux virtual machine using the Secure Shell (SSH) client. Enter "dockeradmin" as the User Name. Now we need to choose an authentication type. We can choose between Password and SSH public key. The most common way of connecting to remote Linux

servers is SSH. The Azure Portal also enables us to enter a password, but we want to follow some standard security best practices and create a "password-less" virtual machine.

We now open the .pub file that we have created earlier in a text editor like TextEdit, and copy the key into the clipboard (you can delete username@machine at the end of the key).

The last step of the VM generation process is to paste the key into the SSH Public Key text field in the Create VM blade.

We can leave the defaults for pricing tier, optional configuration, resource group, and subscription option. The only thing we should change is the location option. We want to make sure to choose a location close to our location to avoid latency issues when accessing the virtual machine. Finally we can click the "Create" button to start the creation of the Azure VM. The portal homepage will appear and provide a status of the creation as shown in Figure 2.8.

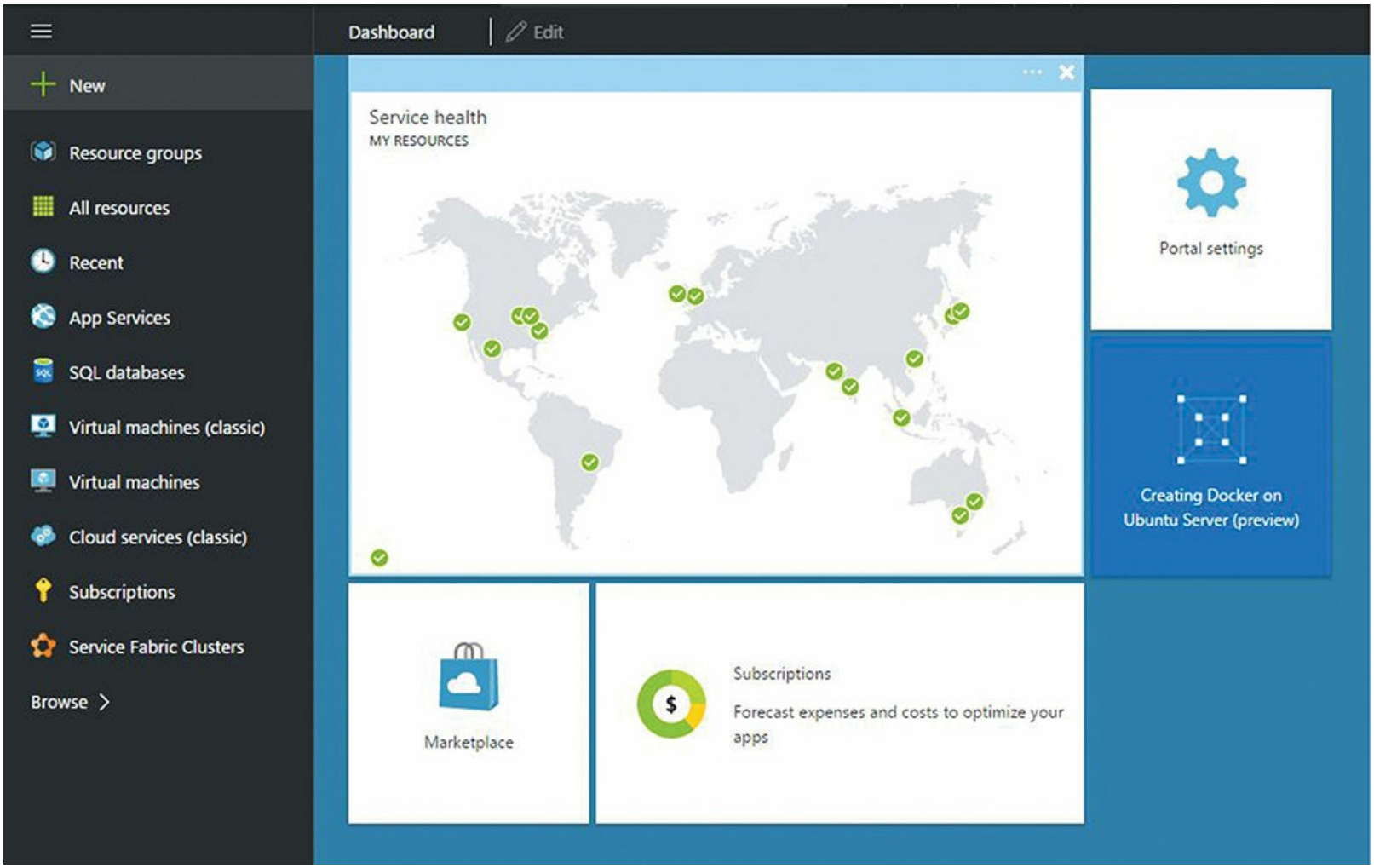

FIGURE 2.8: Status of Azure VM creation

Once the Azure VM is up and running, usually within five minutes or less, the portal opens the Virtual Machine blade that contains useful information such as the status of the machine, DNS name, location, virtual IP address and so on. From this blade, we need to get the full DNS name of the Azure VM. The Azure Portal appends a random string to the host name to make sure the DNS name is unique. Figure 2.9 shows the portal blade with the DNS name of our deployment being dockerhost-3udvgzn4.cloudapp.net.

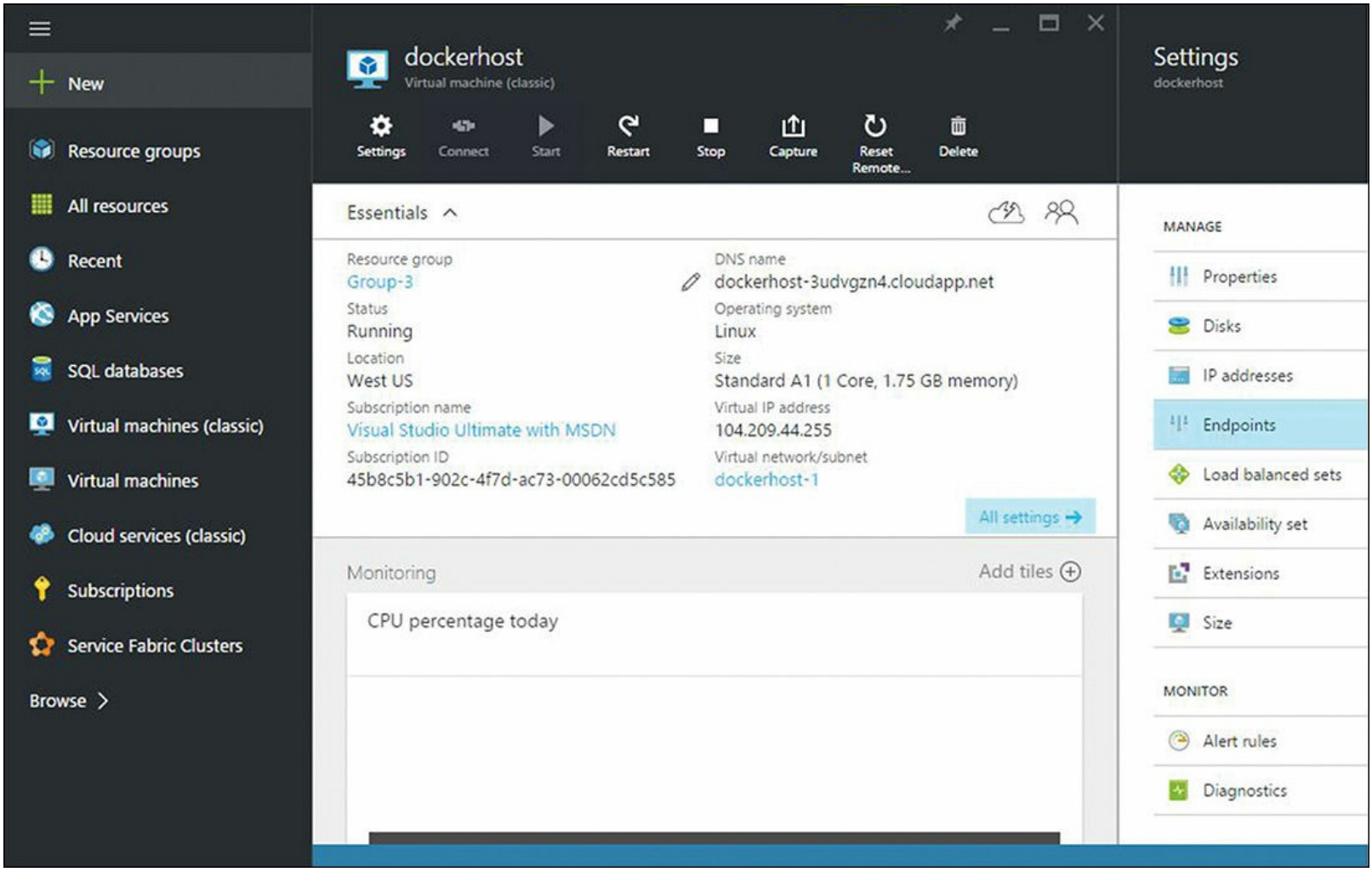

FIGURE 2.9: Blade with Azure VM information

Connecting to the VM Using SSH and Git Bash on Windows

Now we can connect to the Azure VM by entering the following command into the Git Bash shell:

Click here to view code image

```
$ ssh -i ./dockerhostkey dockeradmin@dockerhost-3udvgzn4.cloudapp.net
-p 22
```

> **Note**
>
> We use the identity file parameter "–i" to specify the path to our private key file. We use the port parameter "–p," which is optional when SSH is enabled on port 22. Azure sometimes provisions the Azure VM with a random SSH port. We can find the SSH information at the end of the Properties blade of our Azure VM.

Once we hit "Enter," we are asked for the passphrase that we entered when creating the public key (if you chose to enter a passphrase during creation). After a successful authentication, our Git Bash shell should look similar to Figure 2.10.

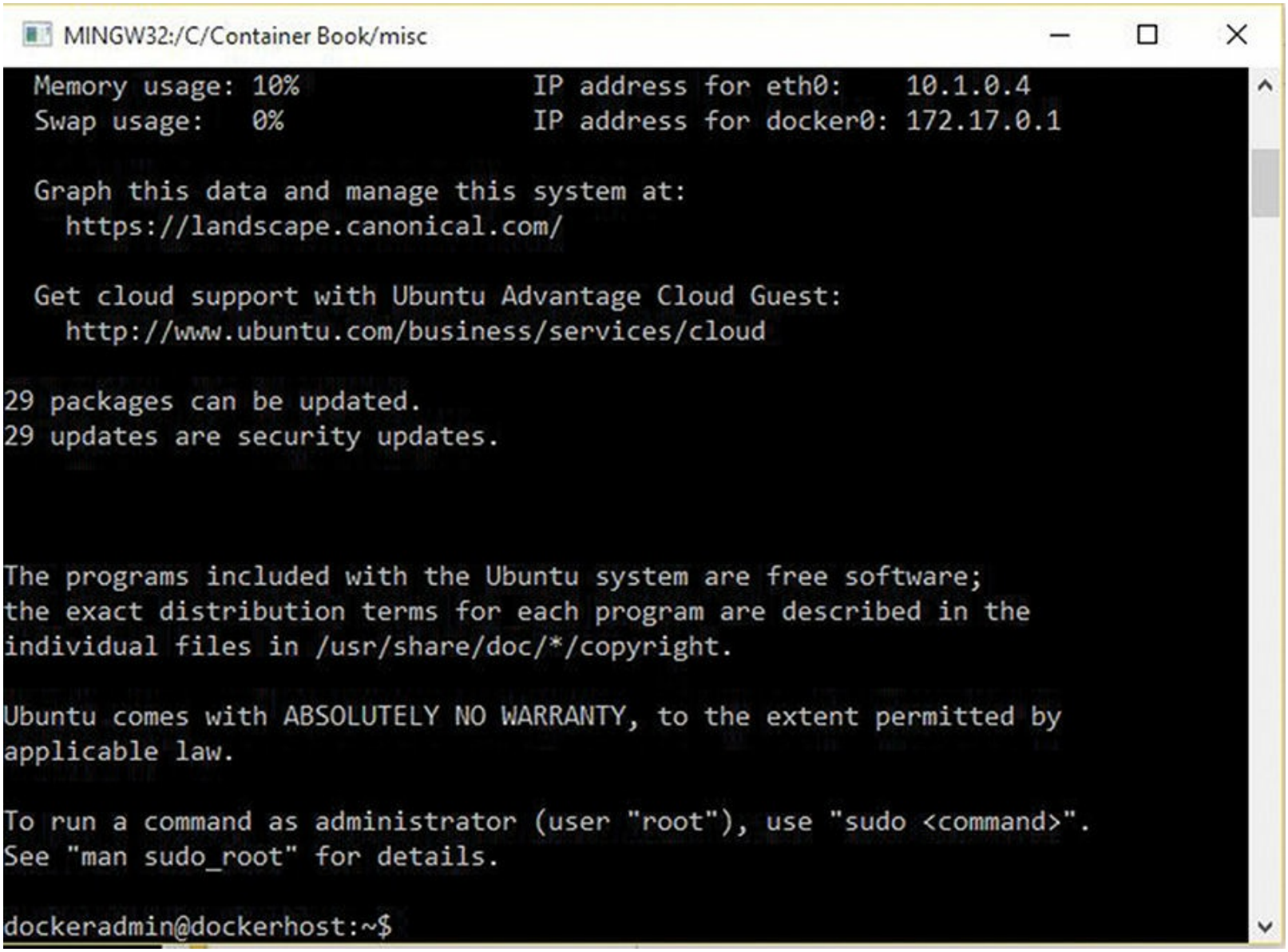

FIGURE 2.10: Git Bash shell connected to Azure VM

Connecting to the VM Using SSH and Git Bash on Mac OS X

Open the terminal and enter the following command:

Click here to view code image

```
$ ssh -i ./dockerhostkey dockeradmin@dockerhost-3udvgzn4.cloudapp.net
-p 22
```

The SSH command is identical to the one used with Git Bash on Windows. Figure 2.11 shows the terminal window after successful connection to the Azure VM.

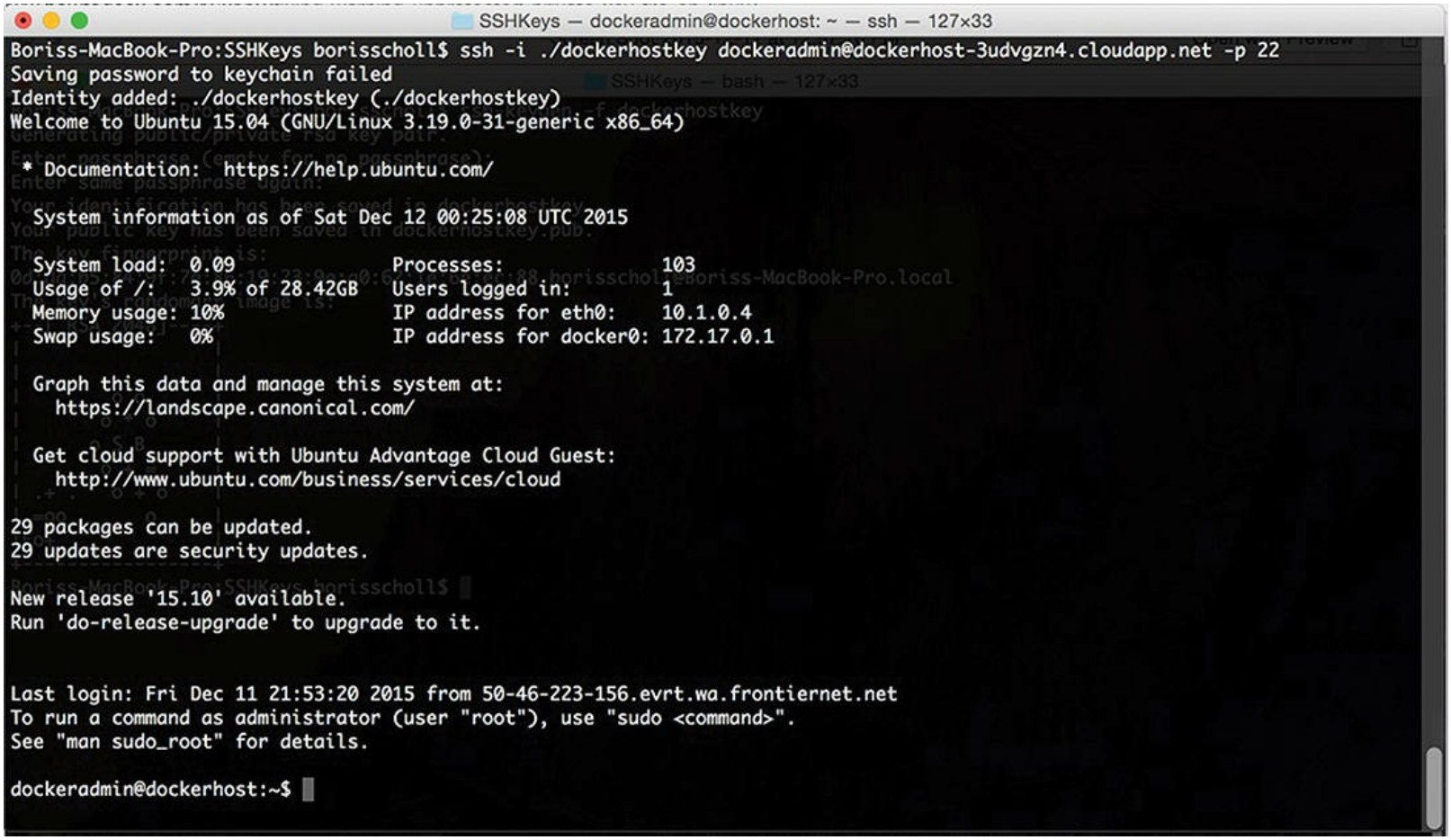

FIGURE 2.11: MAC OS terminal session connected to the Azure VM

Before we move on and get to the Docker basics, let's recap what we have done during the last couple of steps.

- We used the Azure Portal to provision a virtual machine based on the "Docker on Unbuntu Server (preview)" image.
- We created an SSH public key on Windows (using Git Bash) and on Mac OS X (using the terminal).
- We connected to the VM using SSH.

These steps show that it is very easy to set up an Azure virtual machine with the Docker daemon on it and connect to it.

Docker Container Basics

Now that we are connected to our Azure VM we can start exploring Docker.

Docker Info

To make sure that the extension has successfully installed Docker on the machine, we can check whether Docker is installed by typing

```
docker info
```

as shown in Figure 2.12.

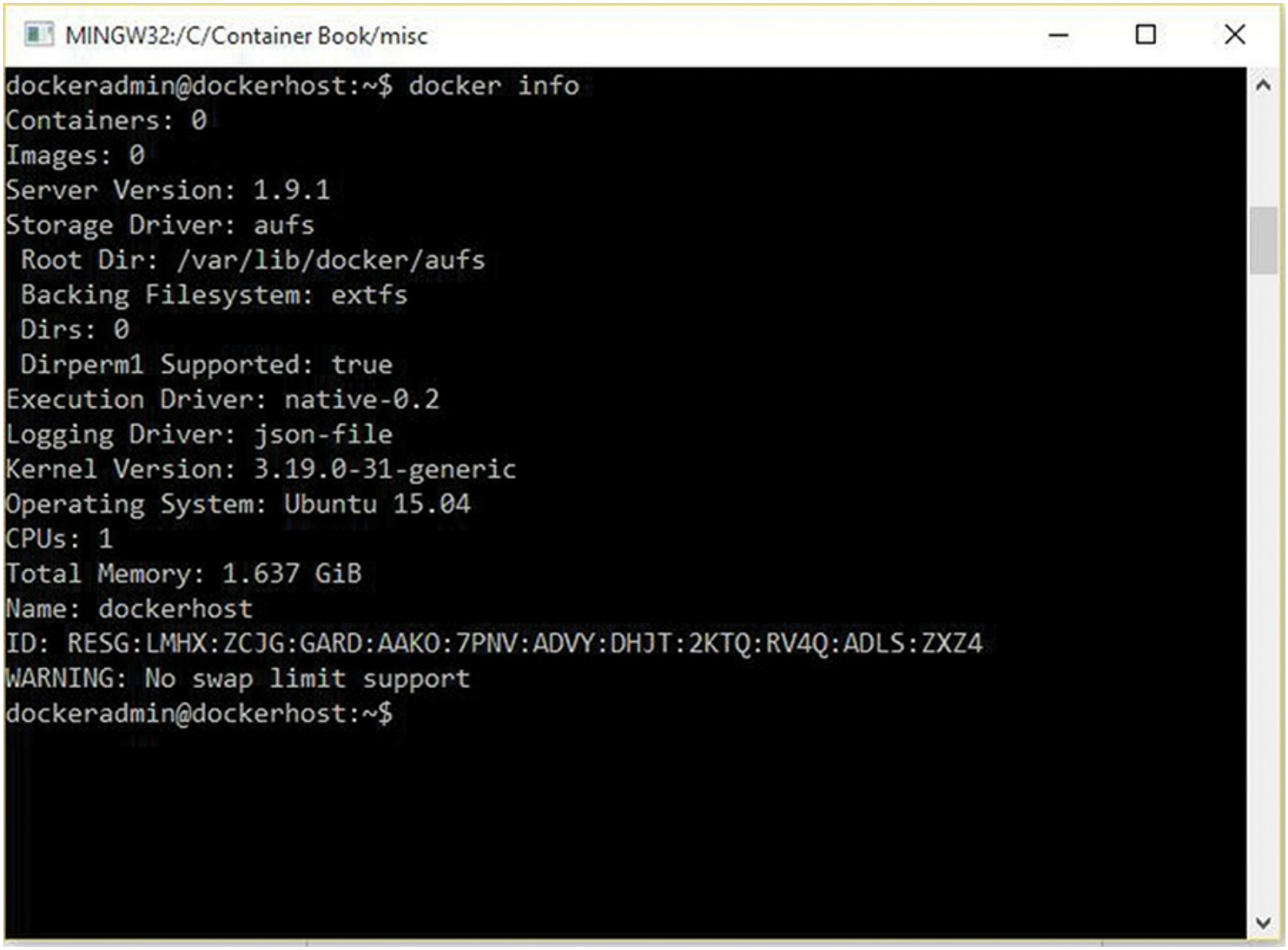

FIGURE 2.12: Docker info output

> **Note**
>
> The following commands are executed on the Docker host via SSH. Therefore, they are exactly the same whether you are connecting to the Docker host from a Windows or Mac OS X machine.

The first two lines indicate that we do not have any containers or images yet, as they show Containers: 0 and Images: 0.

Figure 2.13 shows a logical view of the current state and components of our Azure VM after the successful installation of Docker.

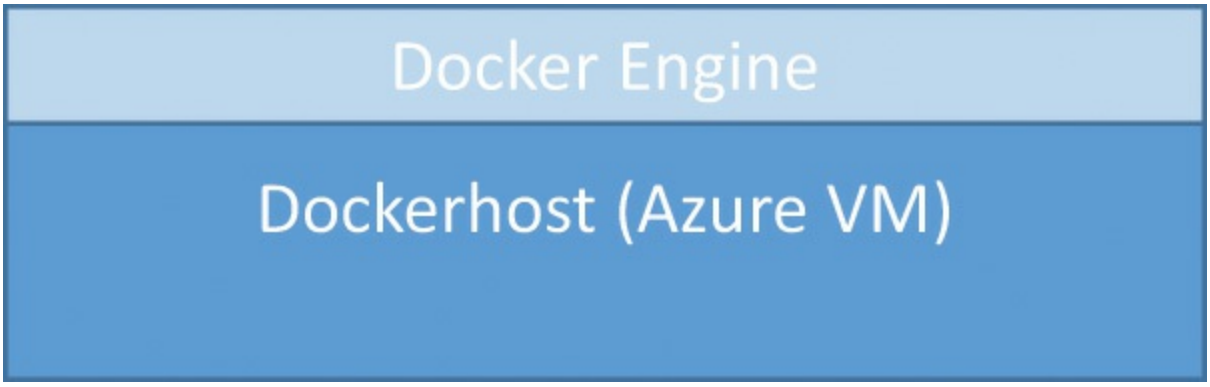

FIGURE 2.13: Azure VM state after provisioning

Now we can create our first container. As in many other examples, we start with a simple scenario. We want to create a container that hosts a simple web site. As a first step, we need a Docker image. We can think of an image as a template for containers, which could contain an operating system such as Ubuntu, web servers, databases, and applications. Later in this chapter, we will learn how to create our own images, but for now we will start with an existing image.

Docker has the notion of image repositories. The public place for Docker repositories is Docker Hub (https://hub.docker.com). Docker Hub can host public and private repositories, as well as official repositories which are always public. The official repositories contain certified images from vendors such as Microsoft or Canonical, and can be consumed by everyone. Private repositories are only accessible to authenticated users of those repositories. Typical scenarios for private repositories are companies that do not want to share their images with others for security reasons, or companies who are working on new services or applications that should not be publicly known.

Docker Pull and Docker Search

To host a simple web application in a Docker container we need a web server. For this exercise, we'll use NGINX. To create a container that is running NGINX we need a Docker image that contains NGINX.

Get the NGINX base image from Docker Hub by entering the following command (you might have to log in to Docker Hub from the CLI of the Docker host before you can pull the image):

```
docker pull nginx
```

The Docker command line also supports searching Docker Hub. The following command returns all images that have NGINX included.

```
docker search nginx
```

Docker Images

During the pull, there is a lot of output in our command window. Running this command caused Docker to check locally for an image called NGINX. If it cannot find it locally, it will pull it from Docker Hub and put it in the local registry on the VM. We can now run

```
docker images
```

to check all the images that are on our VM. Figure 2.14 shows the shell after pulling down the image and running the "docker images" command.

```
MINGW32:/C/Container Book/misc
WARNING: No swap limit support
dockeradmin@dockerhost:~$ docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx

9ee13ca3b908: Pull complete
23cb15b0fcec: Pull complete
62df5e17dafa: Pull complete
d65968c1aa44: Pull complete
f5bb1dddc876: Pull complete
5e5d5160d1c5: Pull complete
cb3657c32fbe: Pull complete
664554bd5e20: Pull complete
2552b0f45ef7: Pull complete
b963fbfcae08: Pull complete
1f6a96635d9e: Pull complete
b4c1ea3cfac2: Pull complete
Digest: sha256:0a8bad8dfc80e38ccdd09c41d4efd2547fa9d6d58d8706431952a2ef312e2034
Status: Downloaded newer image for nginx:latest
dockeradmin@dockerhost:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
nginx               latest              b4c1ea3cfac2        3 days ago
133.9 MB
dockeradmin@dockerhost:~$
```

FIGURE 2.14: NGINX image on dockerhost

Figure 2.15 illustrates the state of the Azure VM after downloading the image from Docker Hub.

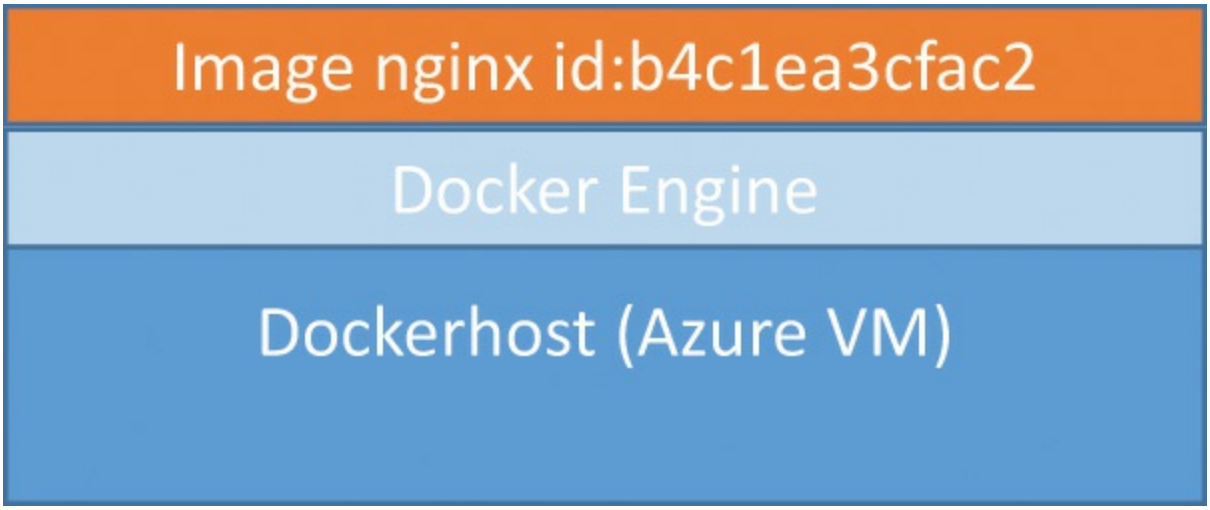

FIGURE 2.15: Logical diagram of NGINX image on dockerhost

Docker Run and Docker PS

Now that we have our image locally, we can start our first container, by entering

Click here to view code image

```
docker run --name webcontainer -p 80:80 -d nginx
```

This command starts a container called “webcontainer” based on the “NGINX” image that you downloaded in the previous section. The parameter “-p” maps ports directly. Port 80 and 443 are exposed by the NGINX image and by using “–p 80:80” we map port 80 on the Docker host to port 80 of the running container. We could have also used the “-P” parameter, which dynamically maps the ports of a container to ports of the host. In Chapter 5, “Service Orchestration and Connectivity,” we’ll see how to use static port mapping because it makes our lives easier when running and exposing multiple containers on virtual machines in a cluster. Finally, the “-d” parameter tells Docker to run that container in the background.

> **Note**
>
> Docker pull is not required to download an image. Docker run will download an image automatically if the image is not found locally.

We can now run the `docker ps` command to check our running container. Figure 2.16 shows the output of the command.

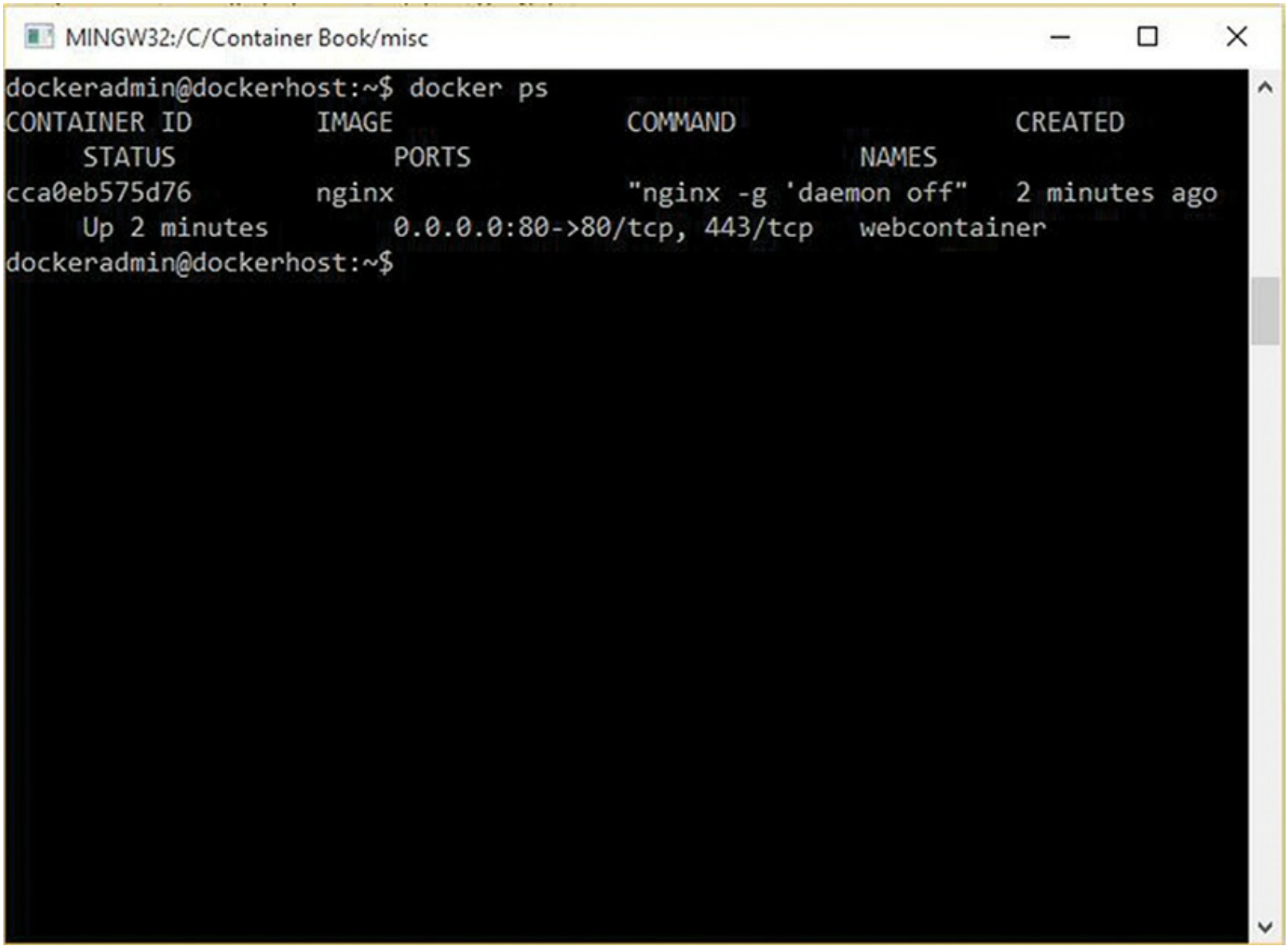

FIGURE 2.16: Output of docker ps

From a topology perspective on the Azure VM, we now have an image (NGINX) and a container (webcontainer) based on that image. Figure 2.17 provides a logical view of our Azure VM running the container "webcontainer."

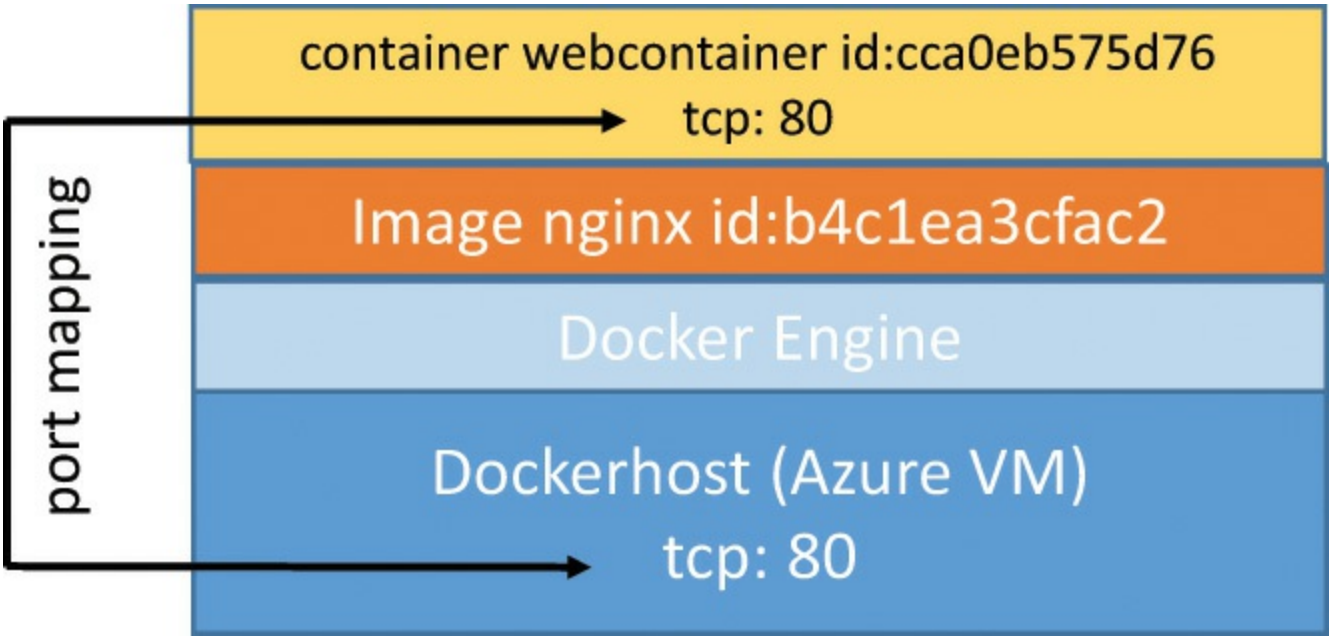

FIGURE 2.17: Logical view of the container running on Azure VM

We can now access the default NGINX web site in our container directly from within the host, for example by executing the following command:

```
curl http://localhost
```

Figure 2.18 shows the welcome web page for NGINX, telling us that the web service is working.

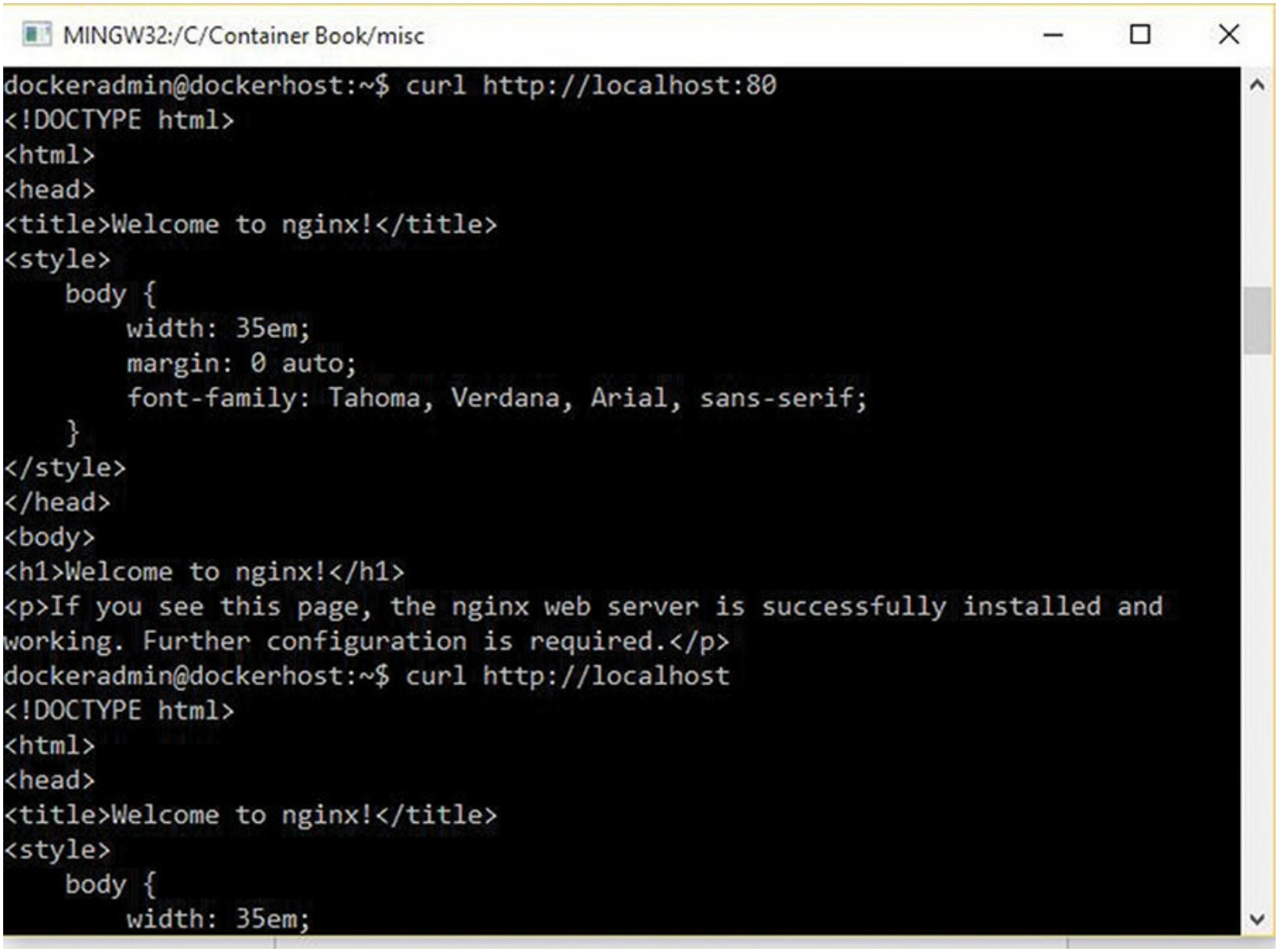

FIGURE 2.18: Working NGINX web server

> **How to Make the Web Site Accessible Through the Internet**
>
> To access the website over the internet we need to add an endpoint to the virtual machine. As with all Azure virtual machine operations, we can use the portal, PowerShell, and CLI to add the endpoint.

We just created our first container based on an image that we pulled from Docker Hub and familiarized ourselves with some basic Docker commands.

Next, we'll look a bit deeper into the Docker basics and learn more about volumes and images, which are important Docker concepts.

Adding Content to a Container Using Volumes

In our scenario, we currently have a running container with the default installation of NGINX in it. This is great but we wanted to host our own web site inside the container. So the question is, how do we handle situations where we need to place custom files inside of a container?

There are a couple of ways of dealing with this. One is to have the container pull in content dynamically and the other one is to make the application and components part of an image.

If we think about a container as an immutable object, we might think that it would make the most sense for all the content to be bundled as part of an image (and therefore part of any containers based on that image). However, several scenarios exist where it makes sense to have a container pull in content dynamically, share data between containers, and have the data independent from the container life cycle.

One very obvious example of this type of scenario is databases, where the data needs to persist in a way in which it is independent from the container life cycle. Think about a scenario where we need to update a container with MySQL to apply security patches. If the data is part of the container image, it would be deleted and that would be a bad thing. Another scenario is the development scenario. A common approach is to have a local reproduction of our application source code on the Azure VM. When we create or start a container, we want to pull in the source code, but we do not want to create a new image every time we make some changes to our source code. Chapter 4, "Setting Up Your Development Environment," covers the development scenario in detail. The solution to both the database and the development scenarios are Docker data volumes.

Docker data volumes are directories that exist within the filesystem of the Docker host that are mounted into running containers. The biggest advantage is that the data persisted in a data volume is independent from the container life cycle, meaning it is not deleted when the container is deleted. From a container life cycle perspective, it is important to understand that data volumes can be shared among multiple containers. The best way to understand data volumes is to create one. Let's recreate the container "webcontainer," using a data volume for our web site content. Before we can create a new one we need to stop and delete the webcontainer that is already running. The following commands will first stop the container and then delete it.

```
docker stop webcontainer
docker rm webcontainer
```

Use the Container ID to Delete a Container

We can also use the first four digits of the container id to delete a container. For this example, the commands for stopping and deleting a container are shown below:

```
docker stop cca0
docker rm cca0
```

We will store the sources for our custom web page in the directory /home/src on the Docker host.

We can use the following commands to create the directory, assuming we are already in the /home directory.

```
mkdir src
cd src
```

Next, let's create a simple HTML page called index.html in the src directory. We can use the nano editor to create that file. Type `nano` and hit "Enter" to open the editor.

The content of the HTML page is very simple and is shown below:

Click here to view code image

```
<html>
    <head>
    </head>
```

```
        <body>
                This is a Hello from a website in a container!
        </body>
    </html>
```

Save the file as “index.html.”

Let’s look at how to mount directories from the Docker host into containers. To mount the host directory we need to execute the following command:

Click here to view code image

```
docker run –name webcontainer –v /home/src:/usr/share/nginx/html:ro
–p 80:80 –d nginx
```

We have already learned that this command will create a container called webcontainer based on the NGINX image and map port 80 on the Docker host to port 80 in the container.

The new part is

Click here to view code image

```
-v /home/src:/usr/share/nginx/html:ro
```

The “-v” parameter mounts the “/home/src” directory created earlier to the “/usr/share/nginx/html” mount point in the container.

> **Note**
>
> The NGINX image uses the default NGINX configuration, so the root directory for the container is /usr/share/nginx/html.

Once the container is up and running, we can check our changes by entering `curl http://localhost`. The output should now look like Figure 2.19.

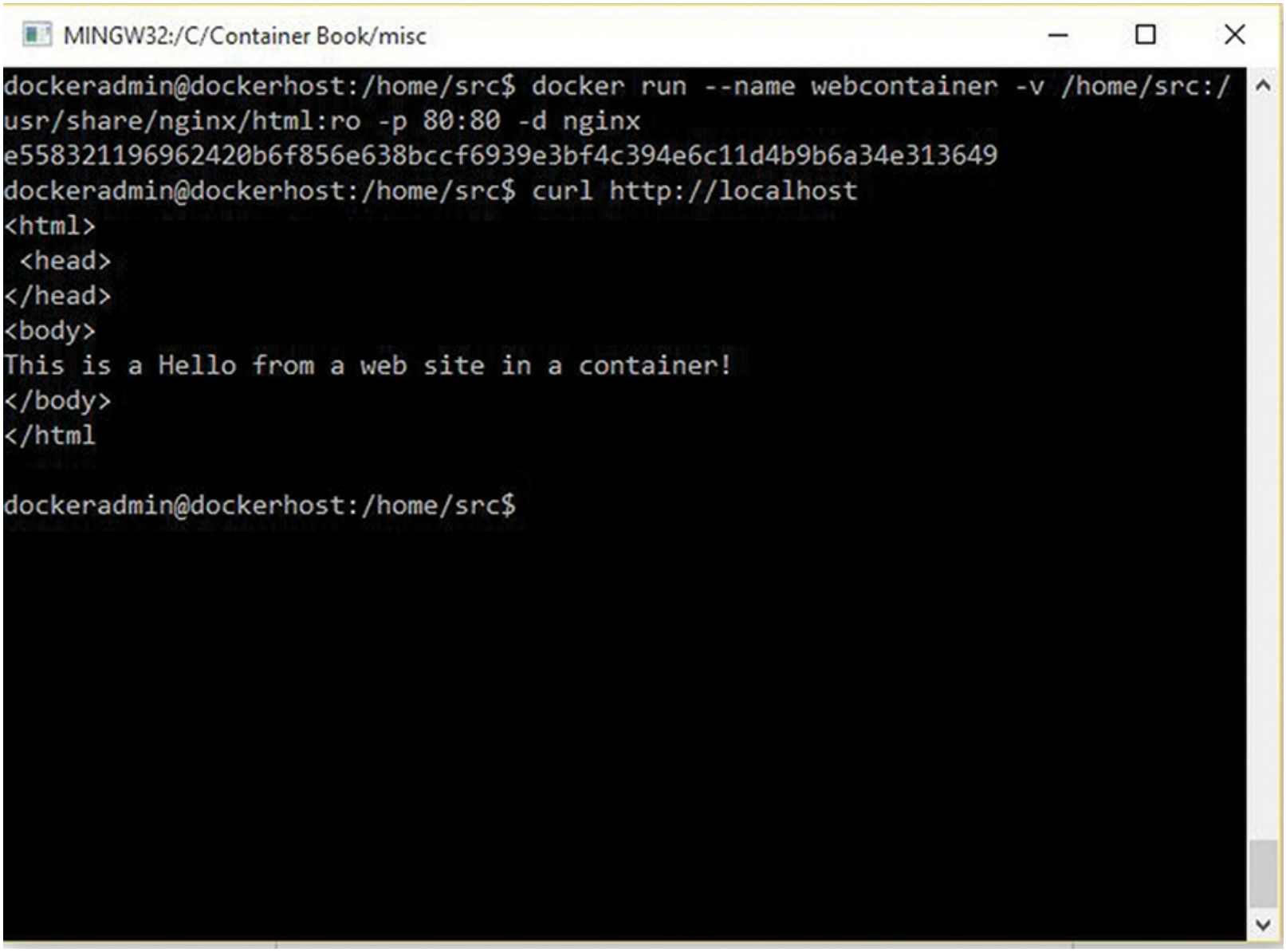

FIGURE 2.19: Web site with custom content in simple webcontainer

Updating and Committing an Image

Another option is to update a container and commit the changes to an image. The first step would be to create a container with a standard input (stdin) stream. The following command creates a container and allocates a pseudo-tty by using the "-t" parameter, and opens a standard input (stdin) stream using the "-i" parameter:

Click here to view code image

```
docker run -t -i nginx /bin/bash.
```

This will create a new container based on the "NGINX" image and drop you into the container's shell, which will look similar to

```
root@67337e2dbcbb:/#
```

Now we can go ahead and install software and make other changes within the container. In our example, we resynchronize the package index files using `apt-get update`.

Once the container is in the state we want it to be, we can exit the container by entering

```
root@67337e2dbcbb:/# exit
```

Finally, we can commit a copy of that container to a new image using the following command from the Docker host:

Click here to view code image

```
docker commit -m "updates applied" -a "Boris Scholl" 67337e2dbcbb
bscholl/nginx:v1
```

Figure 2.20 shows the entire flow of dynamically creating a new image.

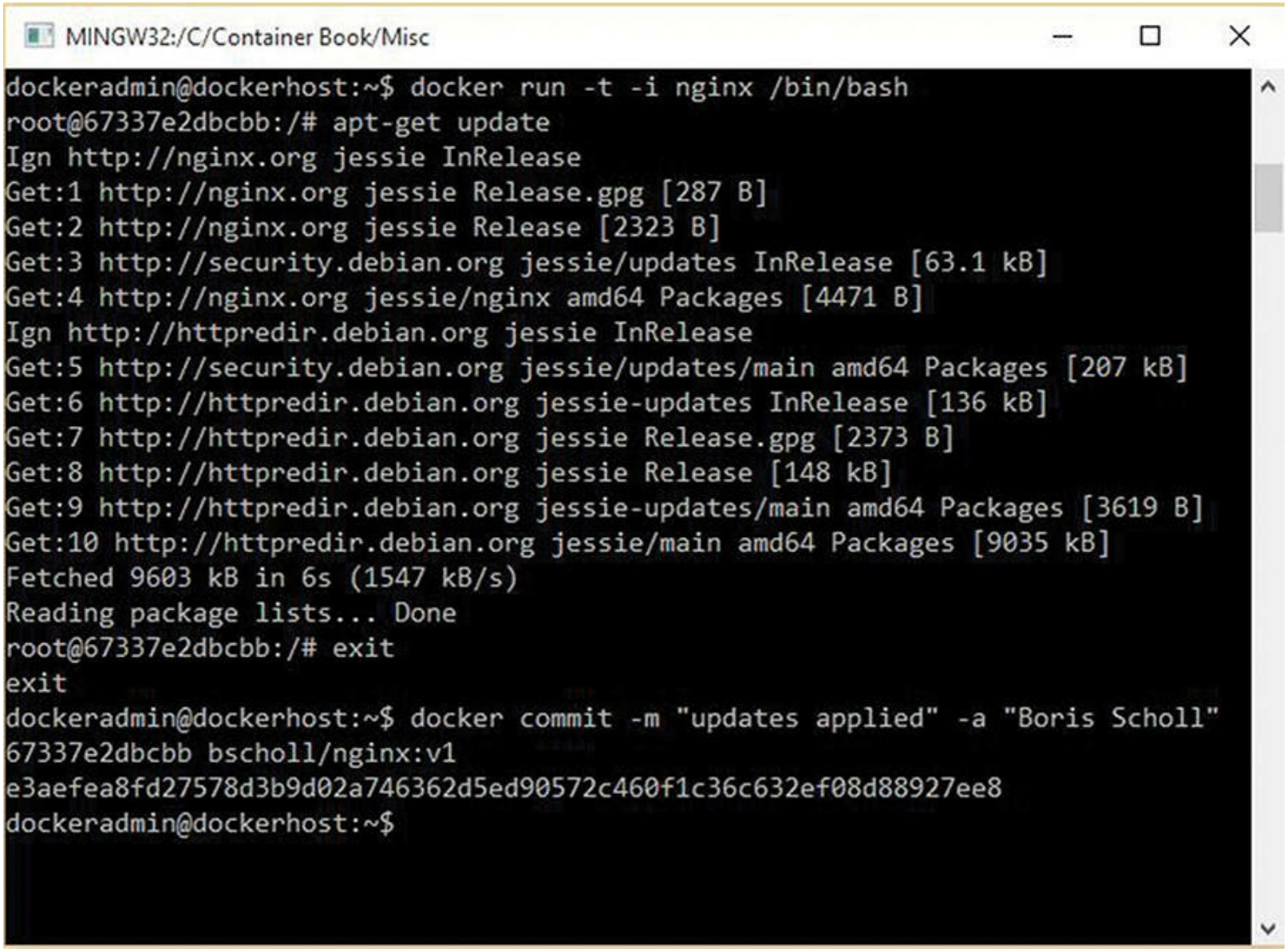

FIGURE 2.20: Create a new Docker image using docker commit

Adding Content to an Image Using a Dockerfile

We can also copy content into a container using a Dockerfile. A Dockerfile is a text file that contains instructions about how to build a Docker image, and as we shall see, it is the more preferred approach.

Let's start looking at the basic Dockerfile structure and syntax by using our NGINX example. The code below shows a Dockerfile that copies the contents of the "web" directory on the Docker host into the directory "/usr/share/nginx/html" in the container.

Click here to view code image

```
#Simple WebSite
```

```
FROM nginx
MAINTAINER Boris Scholl <bscholl@flak.io>
COPY web /usr/share/nginx/html
EXPOSE 80
```

The first line represents a comment as it is prefixed with the "#" sign. The "FROM" instruction tells Docker which image we want to base our new image on. In our case, it is the NGINX image that we pulled down earlier in the chapter. The "MAINTAINER" instruction is to specify who maintains the image. As we will see in a later chapter, that is quite important information when we deal with several teams and many images. The "COPY" instruction tells Docker to copy the contents of the "web" directory on the Azure VM (Docker host) to the directory "/usr/share/nginx/html" in the container. Below is the folder structure on the Azure VM:

```
|-/src
  |-Dockerfile
  |-web
    |-index.html
```

Note

When copying files in the Dockerfile, the path to the local directory is relative to the build context where the Dockerfile is located. For this example, the content to copy is in the "src" directory. The Dockerfile is in the same directory.

Finally, we use the "EXPOSE" instruction to expose port 80. Now that we have the Dockerfile, we can build our image. The following command builds the image, and by using the "-t" parameter, we tag the image with the "customnginx" repository name.

```
docker build -t customnginx .
```

The period (".") at the end of the command tells Docker that the build context is the current directory. The build context is where the Dockerfile is located, and all "COPY" instructions are relative to the build context.

Once the build has been successful, we can run the `docker images` command again to see what images we now have in our local repository. Figure 2.21 shows the output of the `docker build` and `docker images` commands:

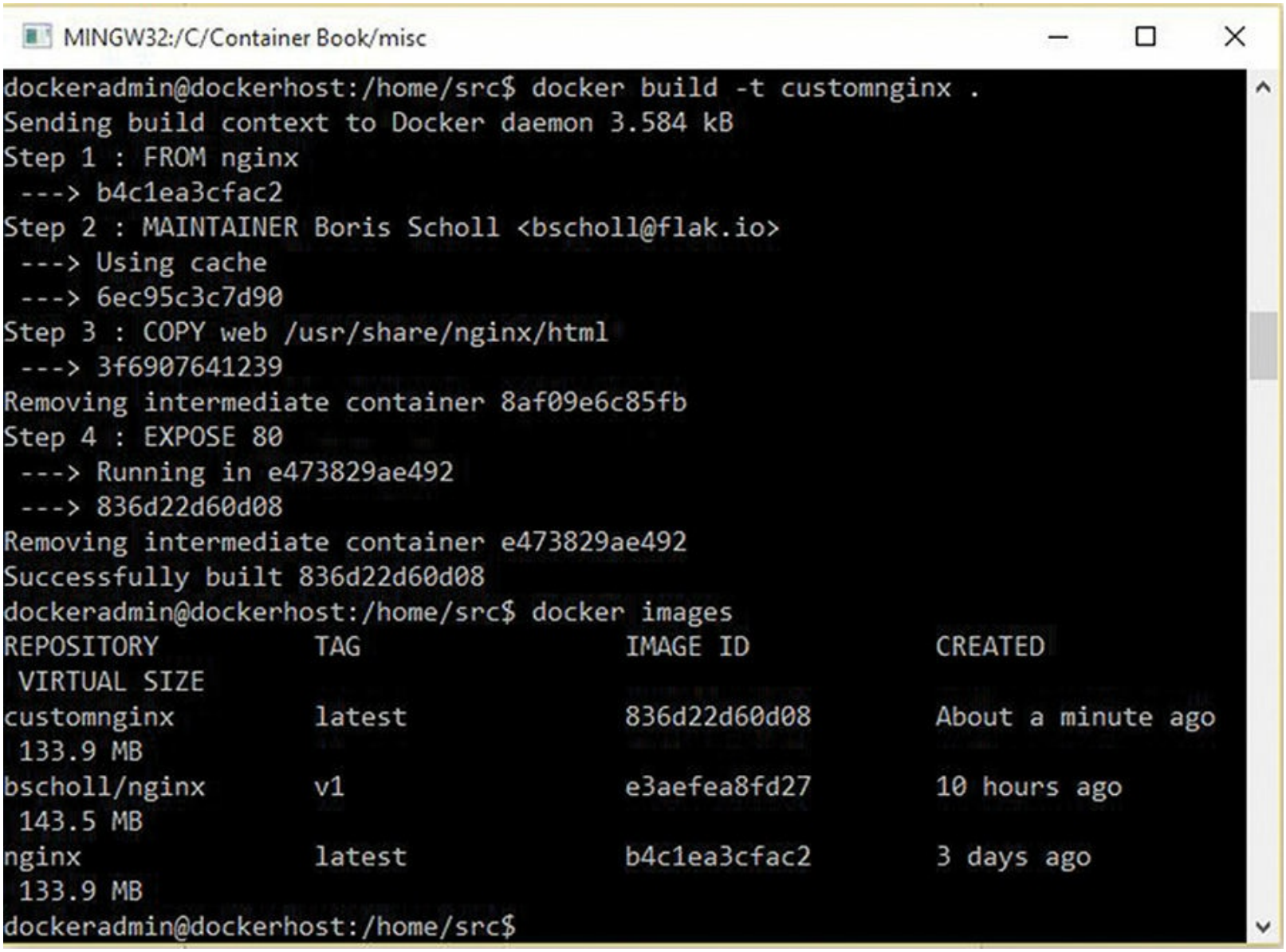

FIGURE 2.21: Docker build and docker images output

As we can see, there are now three images on the Azure VM.

- **nginx:** This is the official NGINX image we pulled from Docker Hub.
- **bscholl/nginx:** This is the image we created using `docker commit`.
- **customnginx:** This is the image we created from the Dockerfile using docker build.

This was just a small example to demonstrate how to create a Docker image using a Dockerfile. Table 2.1 provides a list of the most common instructions to use with Dockerfiles to build an image.

FROM	Sets the base image that we want to build from
MAINTAINER	Sets the author field of the generated image
RUN	Executes a command inside the image; for example, installing a package or cloning a git repo
CMD	Sets default commands to be executed when the container launches. If the ENTRYPOINT instruction is used, values from CMD will be passed to ENTRYPOINT as arguments.
EXPOSE	Exposes a port
ENV	Sets an environment variable
ADD	Copies files, directories or remote file urls from the host to the container
COPY	Copies files or directories from the host to the container. Using COPY is the preferred way of copying files or directories into a container based on the Dockerfile best practices guidance. COPY just uses a basic copy whereas ADD has additional features such as local *.tar extraction.
ENTRYPOINT	Sets the default command to run when the container is launched
VOLUME	The VOLUME command is used to enable access from your container to a directory on the host machine
USER	Sets the user to run the containers from the image
WORKDIR	The WORKDIR directive is used to set where the command defined with CMD is to be executed
ONBUILD	ONBUILD directives in a Dockerfile are executed by images that derive from the image specified in the Dockerfile. Here is an example: Let's say we have a Dockerfile with an ONBUILD instruction to add a software package C to an image. Now we create an image "ImageA" from this Dockerfile. Next we create a new Dockerfile that uses ImageA as base image (FROM ImageA) and builds a new image called "ImageB." When we build "ImageB," Docker will add software package C to ImageB.

TABLE 2.1: Common commands

To finish our Dockerfile exercise, we should test if we can create a new container based on our new image customnginx.

First, we should delete the webcontainer that we have created in our mounting exercise by executing:

```
docker stop webcontainer
docker rm webcontainer
```

Next, we create a new container by executing:

Click here to view code image

```
docker run --name webcontainer -d -p 80:80 customnginx
```

Executing `curl http://localhost` should return the same page as previously shown in Figure 2.19.

In this chapter, we have pulled down the NGINX image from Dockerhub and created two new Docker images. Figure 2.22 shows the logical view of the Azure VM.

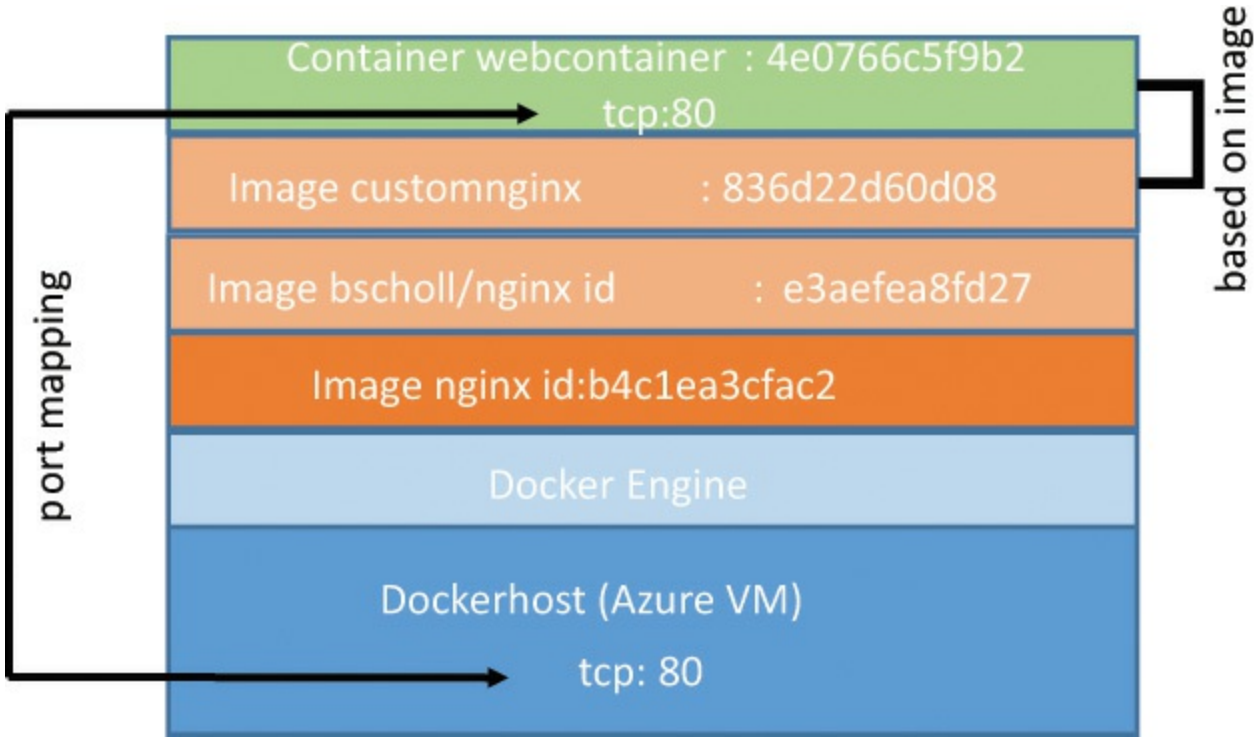

FIGURE 2.22: Logical view of Azure VM

Image Layering

If we look closer at Figure 2.22, we can see another great advantage of Docker. Docker "layers" images on top of each other. In fact, a Docker image is made of filesystems layered on top of each other.

So what does that mean and why is this a good thing? Let's look at the layers of our recently created image by using the

```
docker history customnginx
```

command.

Figure 2.23 shows that the image has 15 layers.

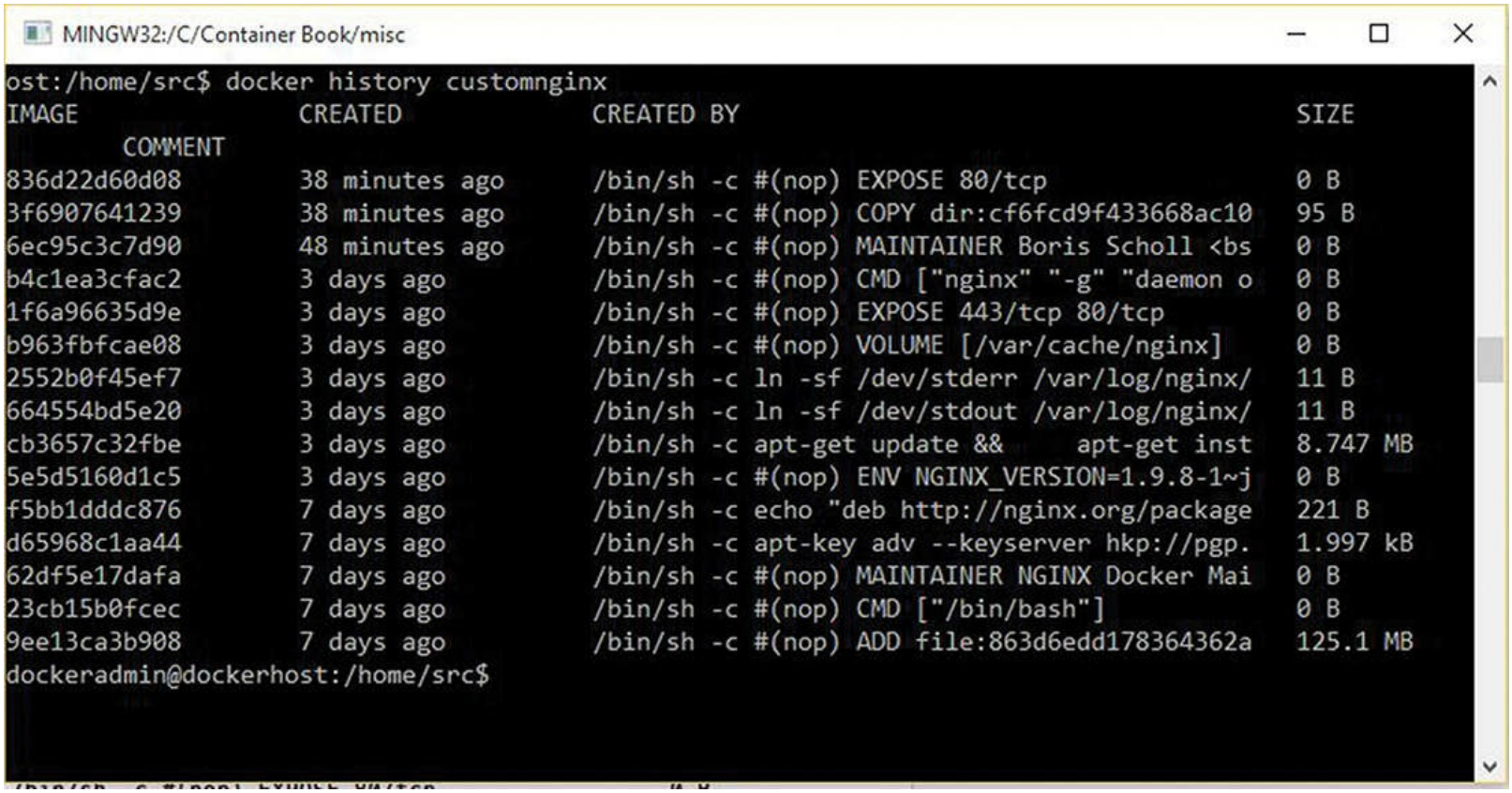

FIGURE 2.23: Layers of the image customnginx

Let's look at the history of the NGINX image by executing

```
docker history nginx.
```

Figure 2.24 shows that the image has 12 layers.

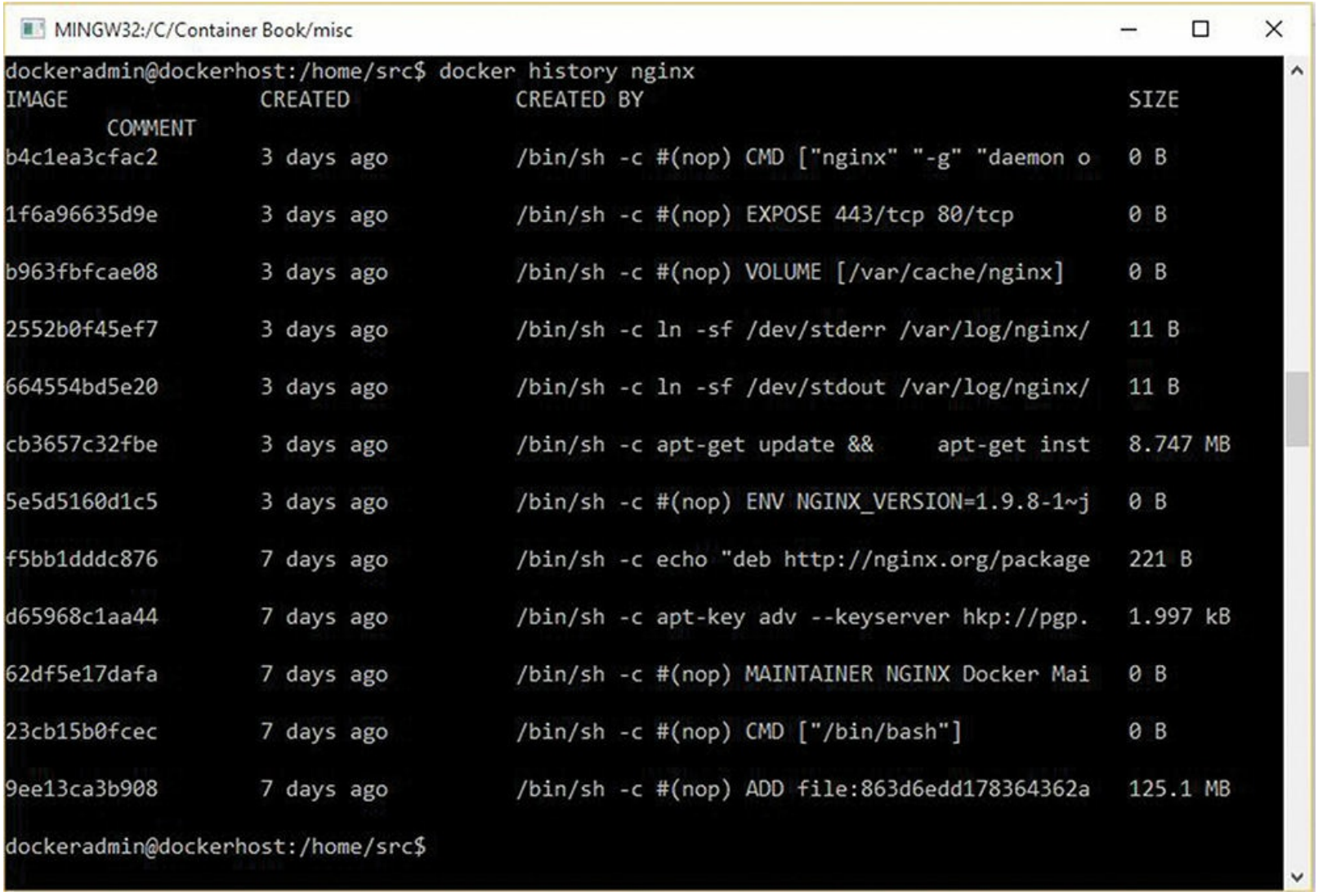

FIGURE 2.24: Layers of image NGINX

If we compare the layers (or intermediate images) of the NGINX image with the layers of the customnginx, we can see that Docker incrementally commits changes to the filesystem, with each change creating a new image layer. The customnginx image has exactly three layers more than the base image. If we look at the description of the three additional layers, we will find the commands used in Dockerfile:

Click here to view code image

```
MAINTAINER Boris Scholl <bscholl@flak.io>
COPY web /usr/share/nginx/html
EXPOSE 80
```

This means that Docker adds a layer for each Dockerfile instruction executed. This comes with many benefits, such as faster builds (as images are smaller) and rollback capabilities. As every image contains all its building steps, we can easily go back to a previous step. We can do this by tagging a certain layer. To tag a layer we can simply use the

```
docker tag imageid
```

command.

For the purpose of this book, we do not need to go into the details of how the various Linux filesystems work and how Docker takes advantage of them. Chapter 4 covers image layers from a development perspective.

If you are interested in advanced reading on that topic, you can check out the Docker layers chapter on https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/.

Viewing Container Logs

Chapter 7 covers monitoring in detail, but there are situations where the container won't start or you want to check if the container was accessed by another container or service at the time of execution. In those cases, we can view the logs of a container by executing

```
docker logs webcontainer.
```

The output is shown below:

Click here to view code image

```
172.17.0.1 - - [12/Dec/2015:17:16:11 +0000] "GET / HTTP/1.1" 200 95
"-" "curl/7.38.0" "-"
172.17.0.1 - - [12/Dec/2015:17:51:55 +0000] "GET / HTTP/1.1" 200 95
"-" "curl/7.38.0" "-"
```

If we wanted to continue to see live updates to the logs as they happen, we can add the "--follow" option to the docker logs command as shown below.

Click here to view code image

```
docker logs --follow webcontainer.
```

The docker logs command also offers the parameters --since, --timestamps, and --tail to filter the logs.

Container Networking

Docker provides rich networking features to provide complete isolation of containers. Docker creates three networks by default when it's installed.

- **Bridge:** This is the default network that all containers are attached to. It is usually called docker0. If we create a container without the `–net` flag, the Docker daemon connects the container to this network. We can see the network by executing `ifconfig` on the Docker host.
- **None:** This instructs the Docker daemon not to attach the container to any part of the Docker host's network stack In this case, we can create our own networking configuration.
- **Host:** Adds a container on the Azure VM's network stack. The network configuration inside the container is identical to the Azure VM.

To choose a network other than "bridge," for example "host," we need to execute the command below:

Click here to view code image

```
docker run –name webcontainer –net=host –d – p 80:80 customnginx
```

In addition to the default networks, the –net parameter also supports the following options:

- **'container:<name|id>':** reuses another container's network stack.
- **'NETWORK':** connects the container to a user-created network using `'docker network create'` command. Docker provides default network drivers for creating a new bridge network or overlay network. We can also create a network plugin or remote network written to

your specifications, but this is beyond the scope of this chapter.

Overlay Network

An overlay network is a network that is built on top of another network. Overlay networks massively simplify container networking and are the way to deal with container networking going forward. In Chapter 5, "Service Orchestration and Connectivity," we discuss clusters, which are collections of multiple Azure VMs. The cluster uses an Azure virtual network (VNET) to connect all the Azure VMs, and an overlay network would be built on top of that VNET. The overlay network requires a valid key-value store service, such as Zookeeper, Consul or Etcd. Chapter 5 also covers key-value stores. Chapter 5 covers how to set up an overlay network for our sample application.

Let's have a closer look at the bridge network as it enables us to link containers, which is a basic concept that we should be aware of. By linking containers, we provide a secure channel for Docker containers to communicate with each other.

Start the first container.

Click here to view code image

```
docker run --name webcontainer -d -p 80:80 customnginx
```

Now we can start a second container and link it to the first one.

Click here to view code image

```
docker run --name webcontainer2 --link webcontainer:weblink -d -p
85:80 customnginx
```

The –link flag uses this format: sourcecontainername:linkaliasname. In this case, the source container is webcontainer and we call the link alias weblink.

Next we enter our running container webcontainer2, to see how Docker set up the link between the containers. We can use the `exec` command as shown below:

```
docker exec -it fcb9 bash
```

fcb9 are the first four digits of the container id for webcontainer2.

Once we are inside the container, we can issue a ping command to the webcontainer. As we can see in Figure 2.25 we can ping the webcontainer by its name.

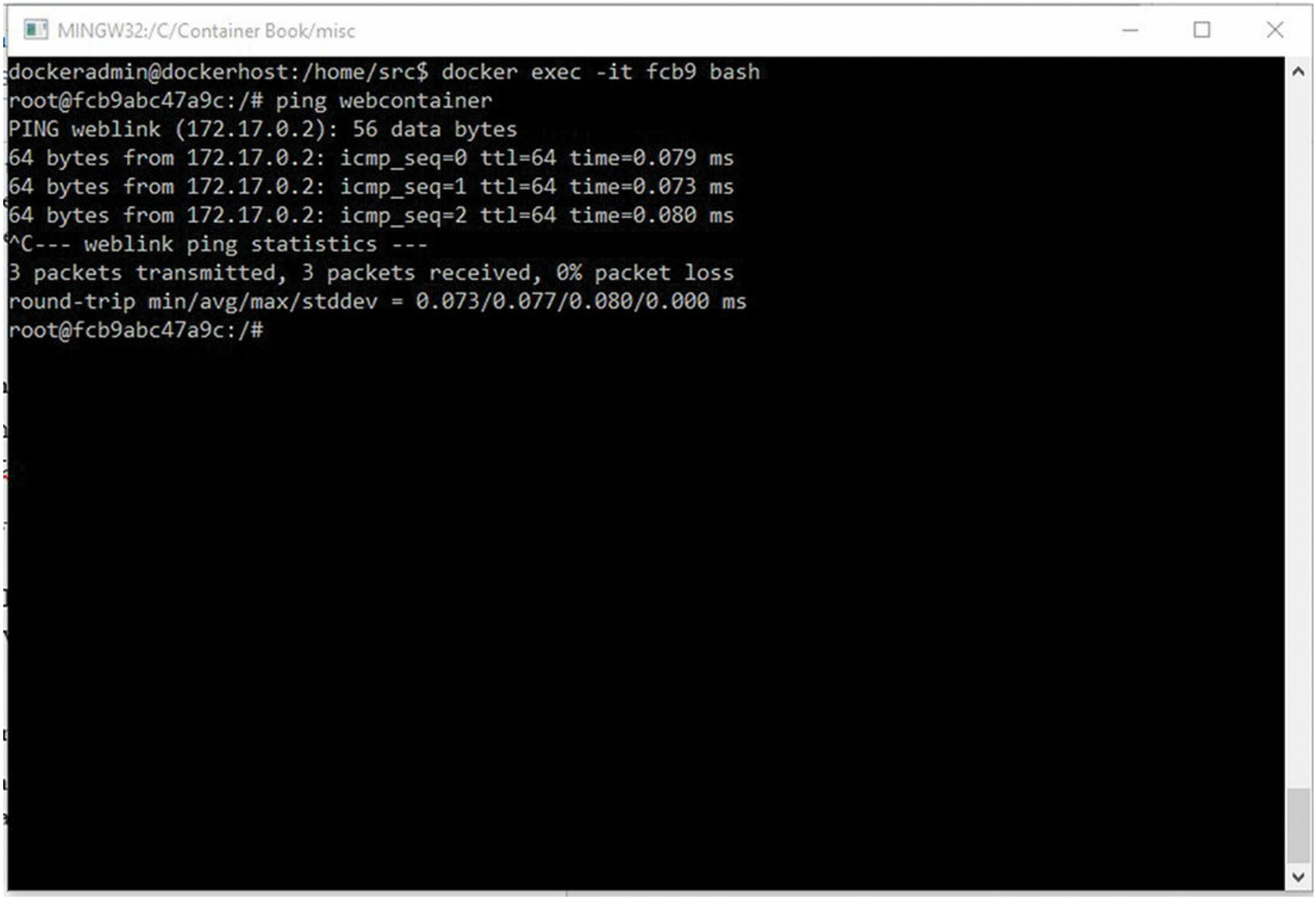

FIGURE 2.25: Pinging the webcontainer

Note that the IP address for webcontainer is 172.17.0.2. During startup, Docker created a host entry in the /etc/hosts file of webcontainer2 with the IP address for webcontainer as shown in Figure 2.26. We can get the host entries by executing

```
more etc/hosts
```

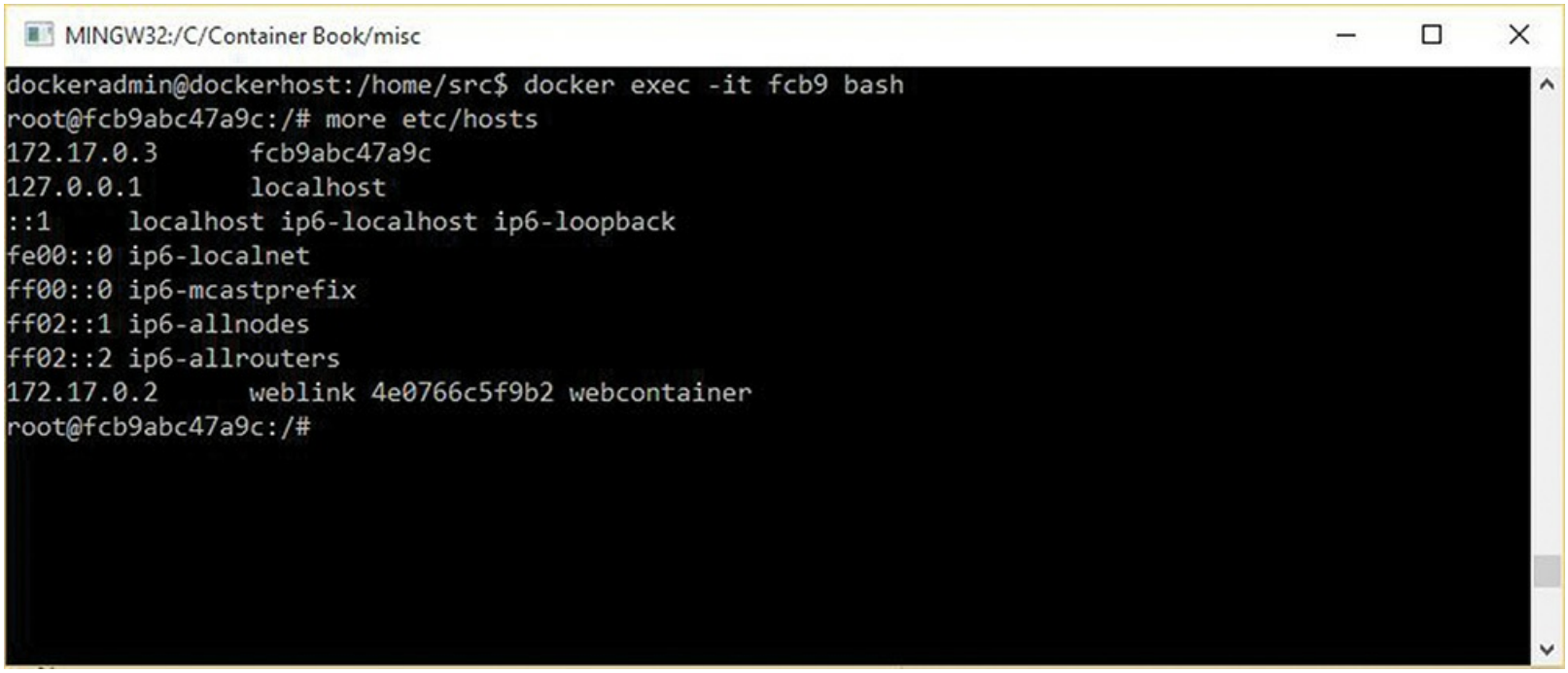

FIGURE 2.26: Linked container entry in /etc/hosts of webcontainer2

In addition to the host entry, Docker also set environment variables during the start of webcontainer2 that hold information about the linked container. If we execute `printenv` we get the output shown in Figure 2.27. The environment variables that start with WEBLINK are the ones containing information about the linked containers.

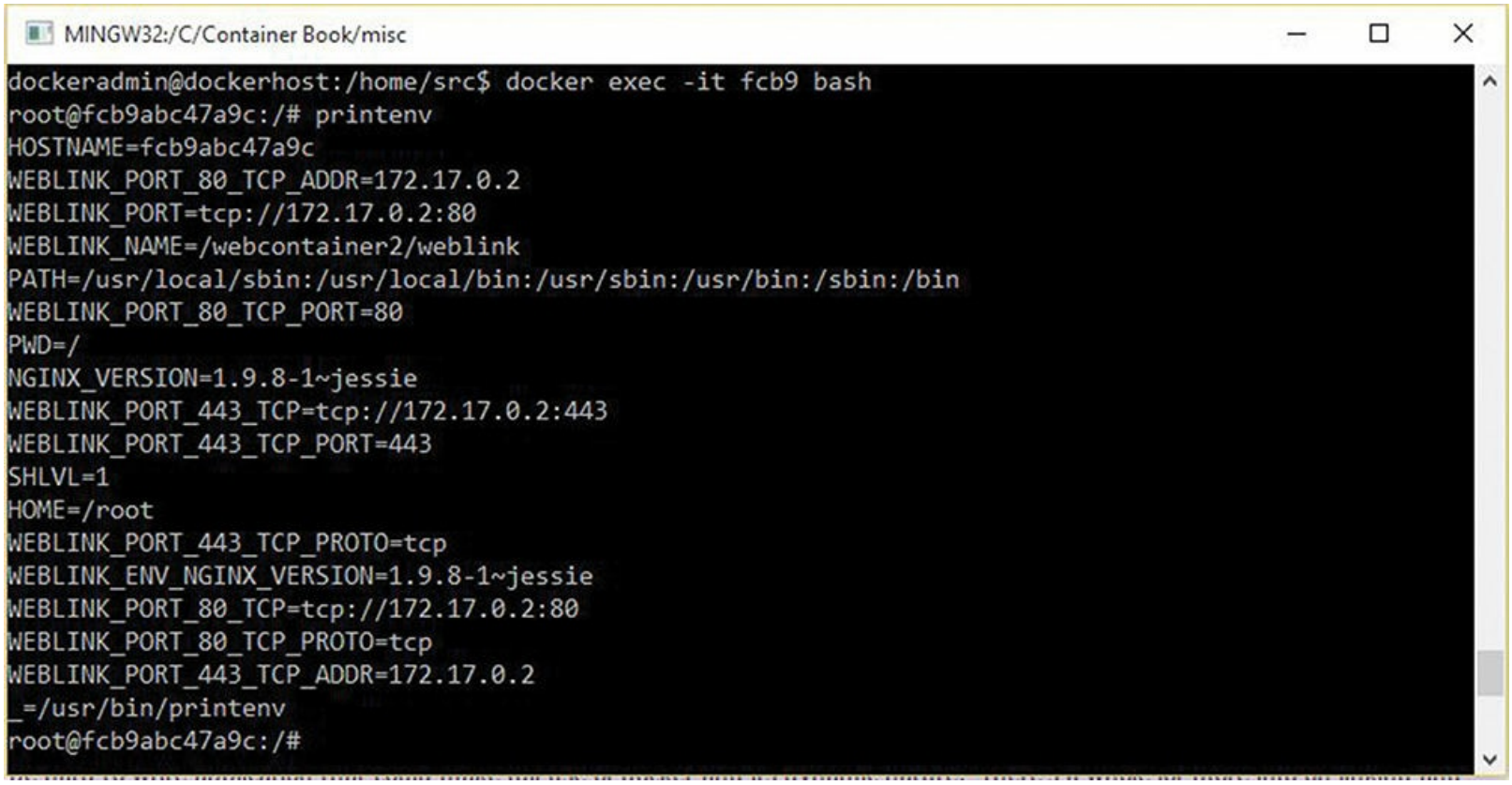

FIGURE 2.27: Environment variables of webcontainer2

For more advanced networking scenarios check out https://docs.docker.com/v1.8/articles/networking/

Environment Variables

Environment variables are critical when we start thinking about abstracting services in containers. Good examples are configuration and connection string information. Environment variables can be set by using the "–e" flag in `docker run`. Below is an example that creates an environment variable "SQL_CONNECTION" and sets the value to staging.

Click here to view code image

```
docker run --name webcontainer2 --link webcontainer:weblink -d -p
85:80 -e SQL_CONNECTION='staging' customnginx
```

Chapters 4 and 6 cover the usage of environment variables in more detail.

Summary

In this chapter, we covered some basic concepts of Docker containers starting from how we can think about the difference between containers, virtual machines, and processes. We also learned how to create virtual machines on Azure that serve as Docker hosts. We went on and learned about basic Docker concepts like images, containers, data volumes, Dockerfiles, and layers. Throughout the chapter, we also familiarized ourselves with basic Docker commands. As we have seen, containers

are a powerful technology that can make us rethink how we build applications in the future.

Over the next chapters, we will use this knowledge to dive deeper into how to work with containers and how to build microservices architectures on top of containers.

3. Designing the Application

In this chapter we will cover some considerations for architecting and designing an application using a microservice architectural style, as well as the paths to a microservices architecture. How do we go about defining the boundaries for the various services and how big should each service be? Before we dive into defining boundaries, let's pause to consider whether or not this is the best approach for the project currently. Sometimes the path to a microservices architecture actually starts with something closer to a monolith.

Most of the successful microservices examples we have to draw experiences from today actually started out as monoliths that evolved into a microservices architecture. That doesn't necessarily mean we can't start a project with a microservices architectural approach, but it's something we will need to carefully consider. If the team doesn't have a lot of experience with this approach, that can cause additional risk. Microservices architecture has a cost that the project might not be ready to assume at the start. We will cover more of these considerations in detail along with some thoughts on defining service boundaries.

> **Note**
>
> As the tools and technologies available in the market for building and managing microservices architecture mature, much of this will change. As the technologies become more advanced, the overhead and complexities in developing and managing a microservices application will be reduced.

Determining Where to Start

Before we jump into designing our microservices and breaking down the business domain, let's first pause to think about whether or not we should start with microservice architecture or plan to eventually work our way to it. There's absolutely nothing wrong with a monolithic architecture for the right project today, and sometimes it's a good place to start. It might even be the case that we can start somewhere in between. A microservices architecture has an up-front cost that can slow the initial release and make it more difficult to get to a minimum viable product quickly, especially in a startup situation. The other thing to consider is the current and future size of the team and the application. If the application is not going to scale to a size that it makes it worthwhile, then we might want to reconsider a microservices approach. Do we have the skills and necessary DevOps practices in place to make it successful? What skills and practices are needed and when can we get them? These are all things we need to consider.

Coarse-Grained Services

A microservices architecture introduces a lot of moving parts, and the initial costs will be higher. It's often better to start with a few coarse-grained, self-contained services, and then decompose them into more fine-grained services as the application matures. With a new project, starting with a monolith or more coarse-grained services can enable us to initially bring a product to market more quickly today. Architects, developers, and operations are often more familiar with the approach, and the tools we use today have been created to work well with this method. As we bring the new application to

market, we can further develop the skills necessary to managing a microservices architecture. The application boundaries will be much more stable and we can better understand where our boundaries should be.

If we do decide to start with something on the monolithic end of the spectrum, there are some considerations to take into account for approaching the design if we plan to transition to a microservices architecture. We could carefully design the application, while paying close attention to the modularity of the application to simplify the migration to microservices. This is great in theory, but in practice, maintaining modularity in a monolith can be challenging. It requires a lot of discipline to build a monolith in a way that it can easily be refactored into a microservices architecture.

> **Note**
>
> Data for each component of the monolith can be deployed to separate schemas in the same database or even separate databases. This can help enforce autonomy in the data store and ease the transition to a microservices architecture.

In some situations, there is absolutely nothing wrong with planning to build a monolith that gets us to market quickly, then replacing it with a microservices architecture and simply discarding the monolith. This requires some planning for how the monolith will eventually be displaced when the time comes.

Depending on the business and technical requirements, as well as the experience and knowledge of the team, we can start at or somewhere between a monolith and fine-grained microservices, as we see in Figure 3.1. Applications with requirements that have more to gain from microservices can start closer to the right on this graph, and teams with less experience can start closer to the left, with plans to further decompose as needed.

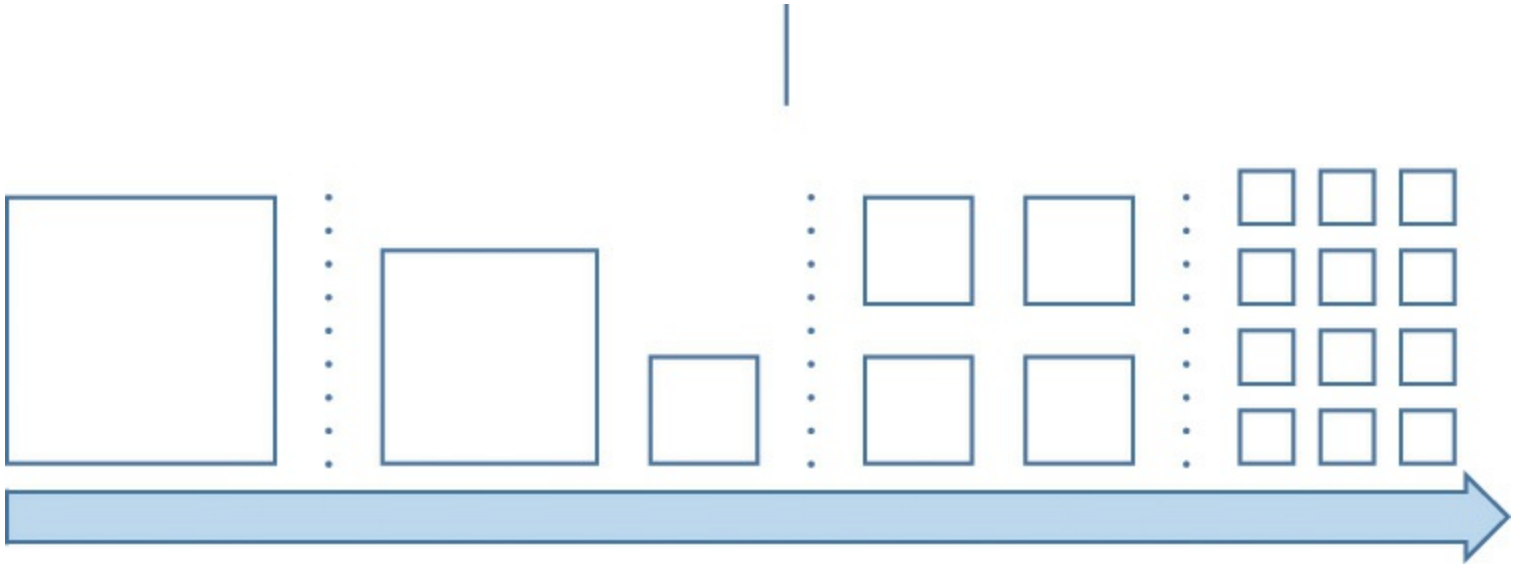

FIGURE 3.1: Evolution of a monolith to a microservices architecture

A valid approach would be to start with fewer coarse-grained services and split them into more fine-grained services over time. We would still break the application up using the same principles and practices for service decomposition and design as we would when approaching a microservices architecture. We can then determine how ready we are with the operational aspects for microservices, and as the team's experience, tools, and processes mature, we can move to finer-grained services. If we are struggling with only a few services, then managing a dozen is going to be

painful.

> **Coarse-Grained Service Decomposition**
>
> When taking an approach to start with more coarse-grained services, use the principles and practices for service decomposition with a microservices architecture. Decompose services around business capability, not application layering.

Starting with Microservices

As we mentioned earlier, there is no reason we cannot start with a monolith or more course-grained services. It could be that we have a very large team that is working on a new large-scale application, and the domain is well-understood. The counter-argument to starting with a monolith is that if we know our end goal is a microservices architecture, then there might be good reason to start there from the beginning.

> **Note**
>
> It's important that the domain is well understood before you begin partitioning it, as refactoring boundaries can be costly and complex.

By starting with a microservices architecture, we can avoid the cost of refactoring later on, and potentially reap the benefits of microservices earlier. We ensure our carefully designed components and boundaries don't become tightly coupled, and based on history they generally will to some degree in a single codebase. The team becomes very familiar with building and managing a microservices-based application from the start. They are able to develop the necessary experience, and build out the necessary infrastructure and tooling to support a microservices architecture. There is no need to worry about whether or not we re-architect it someday, and we avoid some potential technical debt. As the tools and technologies mature, it can be easier to start with this approach.

Once we have made this decision, we will either have a monolith we need to refactor, or a new application we need to build using some combination of coarse- and fine-grained services. Either way, we need to think about approaches to breaking down an application into the parts suitable for a microservices architecture.

Defining Services and Interfaces

Before we can get to work building our services, we need to define what those services are and their interfaces. When we begin decomposing the services, one of the questions that comes to mind is, how big should our services be? Deciding on the number and size of services can be a frustrating task. There are no specific rules for the most optimal size. Some combination of techniques can be used to help determine if we should consider further decomposing a service. The number of files or lines of code can provide some indication that we might want to consider breaking a service up. Another reason might be that a service is providing too many responsibilities, making it hard to reason about, test, and maintain. Team organization, which could require changes to the service partition and the team structure alike, can be a factor. Mixed service types, such as handling authentication and serving web pages, might be another good reason to further decompose a service.

An understanding of the business domain will be important here. The more we know about the capabilities and features of the application, the better we can define the services and their interfaces used to compose that application.

In the end, we want a number of small, loosely coupled services with closely related behavior located together in a service. Loosely coupled services enable us to change one service without needing to change another. It's very important that we are able to update our services independently. We need to take some care when planning integrations between services, so that we do not inadvertently introduce coupling between the services. Accidental coupling could be caused by factors like shared code, rigid protocols, models, shared database, no versioning strategy, or exposing internal implementation details.

Decomposing the Application

When approaching an existing or new application, we will need to determine where the seams are and how to decompose the application into smaller services—among the more challenging and important things to get right in a microservices architecture. We want to make sure to minimize coupling, and that we keep closely related things together. As with good component design, we strive for high cohesion and low coupling in our service design.

Coupling is the degree of interdependence, and low coupling means we have a very small number of interdependencies. This is important for maintaining an independent lifecycle for our services. To keep coupling low, we need to carefully consider how we partition the application, as well as how we integrate the services.

An important thing to consider with regard to a microservices architecture is that when we are discussing high cohesion, we generally mean functional cohesion. Cohesion is the degree by which something is related, and if the degree of cohesion is high, then it's very closely related. Things that have very high cohesion will generally need to change together. If we are making a change to a service we want to be able to make that change in only the one service and release it. We should not have to coordinate changes and releases across multiple services. By keeping closely related things together in a service it can be much easier to accomplish this. This will also tend to reduce chattiness across the services. A key difference with a microservices architectures when compared to traditional SOA is the importance placed on functional over logical cohesion.

> **Refactoring Across Boundaries**
>
> Refactoring across service boundaries is extremely costly and is something we want to avoid if we can. This will likely involve planning, coordination, and possibly data migration as functionality is moved from one service to another.

We need to identify the boundaries in the application that we will use to define the individual services and their interfaces. As we mentioned previously, those boundaries should ensure closely related things are grouped together and unrelated things are someplace else. Depending on the size and complexity of the application this can be a matter of identifying the nouns and verbs used in the application and grouping them.

We can use Domain-Driven Design (DDD) concepts to help us define the boundaries within our application that we will use to break it down in to individual services. A useful concept in DDD is

the bounded context. The context represents a specific responsibility of the application which can be used in decomposing and organizing the problem space. It has a very well-defined boundary which is our service interface, or API.

> **Bounded Context**
>
> A bounded context essentially defines a specific responsibility with an explicit boundary. The specific responsibility is defined in terms of "things" or models within some context. A bounded context also has an explicit interface which defines the interactions and models to share with another context.

When identifying bounded contexts in our domain, think about the business capabilities and terminology. Both will be used to identify and validate the bounded contexts in the domain. A deep dive into Domain-Driven Design (DDD) is out of the scope of this book, but there are a number of fantastic books in the market that cover this in great depth. I recommend "Domain-Driven Design" by Eric Evans and "Patterns, Principles, and Practices of Domain-Driven Design" by Scott Millet and Nick Tune. Defining these boundaries in a large complex domain can be challenging, especially if the domain is not well understood.

We can further partition components within a bounded context into their own services and still share a database. The partitioning can be to meet some nonfunctional requirements like scalability of the application or the need to use a different technology for a feature we need to implement. For example, we might have decided our product catalog will be a single service, and then we realized the search functionality has much different resource and scale requirements than the rest of the application. We can decide to further partition that feature into its own individual service for reasons of security, availability, management, or deployment.

Service Design

When building a service, we want to ensure our service does not become coupled to another team's service, requiring a coordinated release. We want to maintain our independence. We also want to ensure we are not breaking our consumer when deploying updates, including breaking changes. To achieve this, we will need to carefully design the interfaces, tests, and versioning strategies, and document our services while we do so.

When defining the interfaces, we need to ensure we are not exposing unnecessary information in the model or internals of the services. We cannot make any assumptions of how the data being returned is used, and removing a property or changing the name of an internal property that is inadvertently exposed in the API can break a consumer. We need to be careful not to expose more than what is needed. It's easier to add to the model that's returned than it is to remove or change what is returned.

Integration tests can be used when releasing an update to identify any potential breaking changes. One of the challenges with testing our services is that we might not be able to test with the actual versions that will be used in production. Consumers of our service are constantly evolving their services, and we can have dependencies on services that have dependencies on others. We can use consumer-driven contracts, mocks, and stubs for testing consumer services and service dependencies. This topic is covered in Chapter 6, "DevOps and Continuous Delivery."

There will come a time when we need to introduce breaking changes to the consumer and when we

do this, having a versioning strategy in place across the services will be important.

> **A Common Versioning Approach**
>
> A common versioning strategy across all services would be recommended. Service owners have a lot of technology freedoms, but agreeing to a common versioning strategy as well as a common logging format will ease management and service consumption across the organization.

There are a number of different approaches to versioning services. We could put a version in the header, query string, or simply run multiple versions of our service in parallel. If we are deploying multiple versions in parallel, be aware of the fact that this will involve maintaining two branches. A high-priority security update might need to be applied to multiple versions of a service. The Microsoft Patterns & Practices team has provided some fantastic guidance and best practices for general API design and versioning here: https://azure.microsoft.com/en-us/documentation/articles/best-practices-api-design/.

It's also important that the services are documented. This will not only help services get started consuming the API quickly, but can also provide best practices for consuming and working with the API. The API can include batch features that can be useful for reducing chattiness across services, but if the consumer is not aware these batch features do not help. Swagger (http://swagger.io) is a tool we can use for interactive documentation of our API, as well as client SDK generation and discoverability.

Service to Service Communication

Something else that needs to be considered is how services will communicate with one another. The checkout service might need information from the catalog service, or the checkout service might need to send some information to a notification service. Quite often this is in the form of a REST-based API over HTTP/S using JSON serialization, but this is not always the best choice for certain interactions. There are a number of different protocols, serialization formats, and trade-offs to consider. A quick online search will find a lot of great information available covering the performance and management trade-offs with the various serialization options.

> **Serialization Overhead**
>
> Serialization often represents the largest cost with interservice communications. A number of serialization formats are available with various size, serialization, deserialization, and schema trade-offs. If the cost can be avoided, this can help increase performance of the system. We can sometimes pass data through, from upstream to downstream services, and instead add data to headers.

When designing our microservices architecture, we need to consider how our services will communicate. We want to make sure we can avoid unnecessary communication across services and make the communication as efficient as we possibly can.

Whatever we select for our interface must be a well-defined interface that hopefully does not introduce breaking changes for consumers of a specific version. It's not that we can't change the

interface; the key difference is breaking changes. Two different communications styles used between services are synchronous and asynchronous request/response. This means the services implemented will need to use some kind of versioning strategy for the eventual breaking change to an interface.

Synchronous Request/Response

Many developers are very familiar with the synchronous messaging approach. A client sends a request to a back-end and then waits for a response. This style makes a lot of sense: it's easy to reason about and it's been around for a long time. The implementation details might have evolved over time.

One of the challenges with synchronous messaging at scale is that we can tie up resources waiting for a response. If a thread makes a request, quite often it sits and waits for a response, utilizing precious resources and doing nothing while it waits for the response. Many clients can use asynchronous event-based implementations that enable the thread to continue doing work, and then execute a callback when a response is returned. If we need to make synchronous requests to a service, this is the way to do it. The client does, however, expect a timely response from the service. See Appendix A in this book for best practices when implementing and consuming APIs with ASP.NET. Many of the concepts discussed are transferrable to other languages and frameworks.

Asynchronous Messaging

Another useful approach for inter-service communications in distributed systems is asynchronous messaging. Using an asynchronous messaging approach, the client makes a request by sending a message. The client can then continue doing other things, and if the service is expected to send a response it does this by sending another message back. The client will generally wait for acknowledgment that the message was received and queued, but will not have to wait for the request to be processed with a response. In some cases the client does not need a response or will check back with a service at a later time on the status of the request.

There are a number of benefits to asynchronous messaging, but also some trade-offs and challenges that need to be considered. A great introduction to asynchronous messaging is available here from the Microsoft Patterns and Practices team at https://msdn.microsoft.com/en-us/library/dn589781.aspx. This material is a great primer to help those new to asynchronous messaging concepts.

As we mentioned previously, when designing an application using a microservices approach, there are a number of things to consider with regard to communication style, protocols, and serialization formats used.

Considerations with inter-service communications:

- Select a communication approach that is best suited for the interaction, and if possible use asynchronous messaging.
- Consider protocol and serialization formats and the various trade-offs. Performance is important, but so is interoperability, flexibility, and ease of use.
- Always consider how coupling is affected by any decision, because as our services become coupled we tend to lose many of the benefits of a microservices architecture.
- Always consider caching when making requests on external systems.
- Always implement the resiliency patterns covered in Chapter 1 in the Best Practices subsection

when calling other services.

- When one service takes a dependency on another service, it should not require the dependent service to be available at deployment or impact its ability to start. There should never be a need to deploy and start services in a specific order.
- Consider alternate approaches to reduce overhead. For example, if one service uses information received in a request to call another with additional information there might be no need to reserialize the original message. The original message can be passed with the additional information in the header.

There's a place for both synchronous request/response and asynchronous messaging in today's systems. We really just touched the surface on some inter-service communications concepts in this section. For more information on routing requests to services see the Service Discovery section in Chapter 5. Inter-service communication is an important consideration when designing an application based on microservices architecture.

Monolith to Microservices

We might be starting from an existing monolithic application that we want to move to a microservices architecture. This can even be the result of a decision to start with a monolith that we would eventually move to a microservices architecture. Regardless of how we ended up with a monolith, we now want to refactor this monolithic application into a microservices architecture because the benefits outweigh the costs of moving and managing a microservices application. There are a number of different ways to approach this, and we discuss some of those in this section. We have the benefit that we have a fairly well-established business model, and data to help define our service boundaries.

The challenges we often face when breaking down a monolith into a microservices is the tight coupling within the system that contains all kinds of unrelated code that is intertwined throughout the application. It can be very difficult to untangle the pieces of functionality we wish to break out of the application into separate services.

We should first determine the motivations for refactoring a monolithic application into microservices, as this will affect the approach and priorities. If it's because we want to be able to add new features using a different technology, then maybe we don't need to refactor the monolith, and instead can add the feature alongside the monolith. If there's a feature causing some pain or the rest of the monolith is holding back the capability of implementing some feature, then maybe we need to start with moving that one feature first.

Refactoring a monolith is often a process of breaking out one microservice at a time. Parts of the monolith's functionality are replaced with a microservice, and over time the entire monolith has been completely decomposed into a microservices architecture. As the monolith is reduced in size, it will eventually become easier to work through those last few challenging items at the end. Where to start and what to slice off first is something we need to think about very carefully.

Collect Data

Adequate logging and telemetry from a monolith can be extremely useful information when approaching the task of breaking it down into a microservices architecture. This information can be especially useful when we need to partition for scale, or with

identifying coupling in a data store. We might want to ensure our monolith has instruments to gather additional information that will help identify the boundaries for its decomposition into microservices.

We can start by identifying the business capabilities and bounded contexts within the application's domain, then begin analyzing the code. We can find seams in the code, making it easier to decouple one feature easier than another. Dependency tools and profilers can be useful in better understanding the structure and behavior of the application. A feature might exist that has a particular need that is satisfied by a microservices architecture today, like the need to release very quickly; or maybe this feature is fragile and breaks often because of releases to the monolith. It might even be that we want to start with easier-to-partition features and experiment with microservices. Below is a list of things we need to consider and think about when we approach partitioning a monolith.

Considerations for partitioning and prioritization:

- **Rate of changes:** Features that are changing and need to be released often, or those that are very stable and never change.
- **Scale:** Features that require very different scale requirements than the rest of the application.
- **Technology:** A feature can leverage a new technology, making it a good candidate to be partitioned out of the monolith.
- **Organizational structure:** A team working on a feature could be located in a different region.
- **Ease:** There can be some seams and features in the monolith that are easier to partition out and experiment with.
- **Security:** There can be features in the application that deal with very sensitive information and require additional security.
- **Availability:** A feature can have different availability requirements and it can be easier to meet these requirements by breaking it out into its own service. This enables us to more effectively isolate costs to targeted areas of the solution.
- **Compliance:** Certain aspects of the application can fall under compliance regulations and would benefit from being separated from the rest of the application so that the rest of the application is not subject to the compliance rules.

Also consider the fact that we don't have to break out every feature into microservices. It might be better to divide and conquer only if it makes sense for the application. Break up the monolith into some coarse-grained services, and then continue to chip away at one or two large services in parallel.

In addition to splitting out the code into another service, we also have to think about data migration and consider data consistency as we decentralize the data and place it in separate data stores. There will often be some coupling between components and features in the database. The database can be full of challenges like shared data and tables, transactions, foreign key constraints, and reporting needs. We might want to break out the behavior as a first step and enable the monolith and our new service to share the database for some time.

As services are broken out from the monolith, they need to continue to collaborate and integrate with the monolith. A proxy can be placed in front of the monolith to route traffic to the new services as they are broken out. Features partitioned out of the monolith might need to be replaced with client

proxies used to call the new service. Also, where services need to interact with the existing monolithic application, consider adding an anti-corruption layer. The anti-corruption layer is introduced in Domain-Driven Design (DDD), by Eric Evans. This approach creates a façade over the monolithic application and ensures the domain model of the monolith does not corrupt the microservices we are building.

Flak.io e-Commerce Sample

Flak.io is an open-source microservices sample application used for demonstrating microservices concepts and exploring related technologies. In this section we will walk through the sample application design and considerations. The application is small and simple, designed to provide us with enough examples to cover many of the concepts discussed through this book.

Flak.io is an ecommerce application that enables shoppers to browse and purchase products from a catalog. Let's start with a look at some of the flak.io requirements.

Flak.io

Flak.io is an advanced technology reseller that sells unique products online for the modern-day space explorers. Space exploration is growing fast, and the demand for equipment is rapidly growing with it. Flak.io needs to release an ecommerce solution that can scale to handle the expected demands. The online retail market for space exploration equipment is getting very competitive, and flak.io must remain agile and capable of continuous innovation. Flak.io has decided to build their new storefront technology using microservices architecture. The team has some high-level requirements that align well with the benefits of a microservices architecture.

Initially users will simply need to be able to browse a catalog of products and purchase them online. Initially users will not need to log in or create accounts, but eventually this feature will be added. The catalog will include basic browse and search capabilities, enabling a user to find a product by entering search terms and browsing products by category. When viewing products on the site, the user will be displayed a list of related and recommended products based on the analysis of past orders and browsing histories. The user will add products that he or she wishes to purchase to a shopping cart and then make the purchase. The user will then receive an email notification with a receipt and details of the purchase.

Requirements

- **Continuous Innovation:** To keep their positon in the market, the flak.io team needs to be able to experiment with new features and release them into the market immediately.
- **Reduce Mean Time to Recovery (MTTR):** The DevOps team needs to be able to quickly identify and release fixes to the service.
- **Technology Choice:** New technologies are constantly being introduced into the market that can be leveraged in key areas of the application, and it's important that the team is able to quickly adopt them if it makes sense for the business.
- **Optimize resources:** It's important that the application is optimized to reduce cloud infrastructure costs.
- **Application availability:** It's important that some features of the application are highly available.

- **Reduce developer ramp-up:** The team has been growing quickly, and it's important that new developers are able to ramp up and begin contributing immediately.

It's apparent the Flak.io ecommerce application will benefit from a microservices architecture and the costs are worth the return. The team has enough experience in DevOps to service design to handle it, and the application has been decomposed into the following set of services.

Architecture Overview

As we see in Figure 3.2, the application is decomposed into four services.

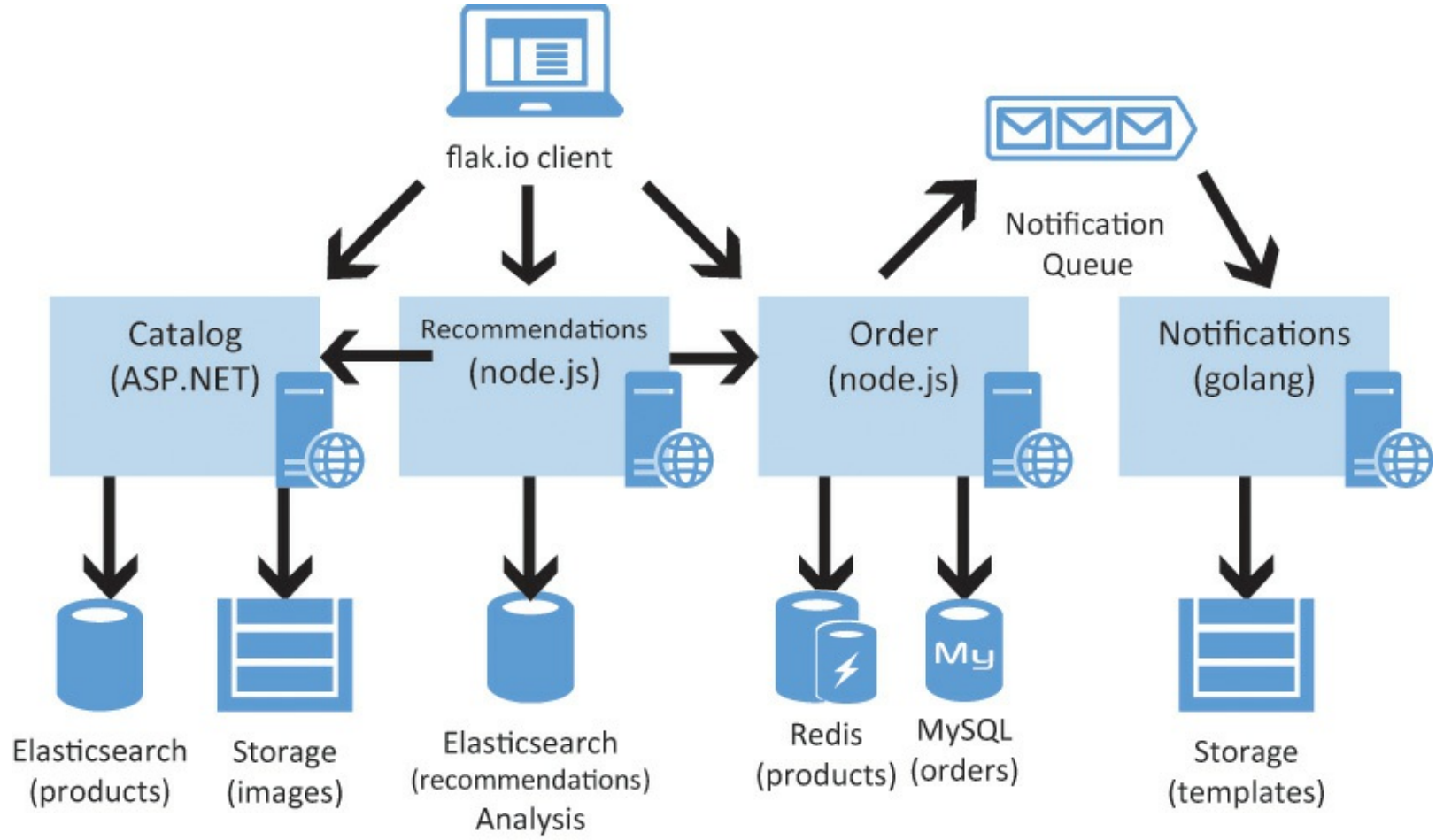

FIGURE 3.2: Flak.io architecture overview diagram

- The catalog service contains categories and product information, and images.
- The order service is responsible for processing orders. You should note that the order service would also have a product model that's different from that in the catalog service, because it belongs to a different context.
- The recommendation service is responsible for analyzing historical information and possibly more real time information to make recommendations.
- The notification service is responsible for sending email notifications, as well as storing and managing email templates.

The application uses a mix of synchronous and asynchronous messaging techniques. The application will be deployed into a cluster using cluster management and orchestration tooling, which will be discussed in more detail in Chapter 5.

Considerations

Considerations

Some things to consider when decomposing an application into individual services:

- Right now the client makes one request to retrieve catalog information and another for recommendations. We might want to reduce chattiness in the client and aggregate the request in the proxy or make one of the services responsible for that. For example, we can send a request to the catalog service which could request recommendations from the recommendation service, or have the edge proxy fan out and call both and return the combined result.
- We might want to further decompose a bounded context around business components, components, or other things. For example, depending on how our application scales, we might want to separate the search function from the other catalog features. We will keep it more coarsely grained until we determine there is a need to further partition.
- Elasticsearch is used in place of other graph databases for providing recommendations for a couple of reasons. Elasticsearch includes an extremely powerful graph API, and because it's used for other areas, we can reduce the number of technologies and share information for best practices.
- We could place the recommendations and products into different indexes of the same Elasticsearch cluster. As long as one service is not permitted to directly access the other service's index, we should be fine and might be able to reduce costs by running multiple clusters.
- It's quite possible that many of these services could be further decomposed into more microservices. For example, the team had considered further decomposing the order service into more services, by moving some of the functionality out to a payment/checkout service. However, the team had determined that refactoring it at a later time would require minimal work, and they were not ready to manage more services or the complexities it involves at this time.
- Domain-Driven Design is an approach and a tool. It's a great tool in this situation, but it's not the only approach to decomposing a domain into separate services.
- At the moment, customers are anonymous and simply enter payment information every time they make a purchase. In the future, a feature will be added to the application that enables customer accounts.

Summary

One of the biggest challenges in designing an application based on a microservices architecture is in defining the boundaries. We can leverage techniques from Domain-Driven Design (DDD) for help defining boundaries. We have also learned that the best path to a microservices architecture can actually start with something a bit more monolithic in nature today. As the experiences and technologies in the industry mature, this can change. Regardless of how we start, we know that a deep understanding of the business domain is necessary. As with any architectural approach, microservices architecture is loaded with trade-offs big and small. Now that we have an application and design, let's move along to building, deploying, and managing it.

4. Setting Up Your Development Environment

In this chapter, we'll discuss the ways we can use Docker in a development environment including local development, Docker image management, local development versus production-ready development, using Docker Compose to manage multiple containers, and common techniques to diagnose Docker issues.

Using Docker for Local Development

There are three common use cases for local Docker development which we'll review here and discuss in more detail in this and later chapters.

Docker for Local Development

For local development, you are using containers as a host for your application, but optimizing for developer productivity and debugging. For example, you typically volume mount your local source code, turn on full tracing or debugging features, and include un-minified JavaScript to simplify debugging.

Docker for Production Validation

For production validation, developers are building a container with the fully optimized code, and disabling development-only features. Instead of volume mounting the source code, production images more typically include Dockerfile instructions to copy any source code directly into the image. This helps developers test that the same exact image they built in development will work exactly the same way in staging and production.

Docker as a Build/Test Host

Some development teams use Docker containers not as a host for their application in production, but rather as an on-demand, isolated environment for compiling and testing. A developer can easily spin up a build inside a container, specify what tests to run, and run them in a container instead of on their local box. One reason this is handy is if a developer needs to install and validate different dependencies like shared libraries or prerelease software that you might not want to install on your local box. After the build compiles and unit tests pass, the container is then discarded. A more common use case for build and test validation with Docker is not on the local machine, but rather as part of a continuous integration workflow, which we'll discuss in Chapter 6, "DevOps and Continuous Delivery."

Developer Configurations

One of the first considerations for developing with Docker is how to set up your development team's configuration. Below are three of the most common usage patterns.

Local Development

In this configuration, all development is done locally on a laptop, typically using containers running inside of a virtual machine. Some developers find they are more productive writing code on their local machine without using Docker, and then, once primary development is done, they test and

integrate their code running in a Docker container against other services.

Local and Cloud

In this configuration, initial development is done locally as described above, but more extensive testing/development is done in a shared environment that has virtual machines running in a public or private cloud that more closely mimics the microservice's real-world production environment.

Cloud Only

In this configuration, all development is done with containers on top of virtual machines in a public or private cloud. Development teams sometimes need to choose this option because their local PCs aren't capable of running virtualization software, or a corporate policy doesn't permit them to run virtualization on their PCs. To work around these restrictions, each developer gets a bare-bones virtual machine hosted in the cloud to run their containers.

Managing Docker Authentication

Once you've chosen how you want to set up your development environment, the next consideration to think about is Docker authentication. Docker typically is set up with a set of certificates to enable remote access of a Docker client using Transport Layer Security (TLS) over port 2376. You can find instructions on how to create certificates and configure the Docker daemon at http://bit.ly/ch4certs. As part of the certificate creation, you will end up with three certificates: cert.pem, key.pem, and ca.pem, all of which should be added to your user folder "~/.docker for Mac" or "C:\Users\<username>\.docker" on Windows as discussed in Chapter 2, "Containers on Azure Basics."

One common issue that will arise is the need to manage multiple certificates. For example, you likely will have different certificates for your local development environment, CI, or staging environments. Similarly, if you're doing development across multiple teams, for example, developing microservices for the finance and human resources departments, each team likely has a different set of certs they use for their environments. One simple way of organizing certs is to have all certs under your .docker machines directory, with each environment as a separate subfolder with its own set of keys. You can then use the DOCKER_CERT_PATH environment variable to set the keys for each environment like the dev environment for finance:

Click here to view code image

```
set DOCKER_CERT_PATH=c:\users\<username>\.docker\machine\machines\
finance-dev
```

or on a Mac:

Click here to view code image

```
export DOCKER_CERT_PATH=~/.docker/machine/machines/finance-dev
```

As each Docker host (virtual machine) needs its own set of keys, you want to make sure you have an easy way to manage a set of authentication keys across multiple developers and multiple virtual machines. Remember not to include your Docker certs in your source control system, as by doing that, you are effectively giving full trust and control to anyone who has access to your source control system. Further, hackers can use automated sniffers that recursively search source control systems for passwords and certificates, so instead of only gaining access to your source control system, hackers

would then be able to fully control your deployment servers.

- For local development, use the built-in Docker keys that are unique to each developer. These are stored in the default directory, located at C:\users\<username>.\docker\machine\machines\default or ~./docker/machine/machines/default on a Mac.
- For a shared development environment, developers typically share a set of Docker keys across the team as developers require the need to directly manage and monitor containers, including viewing logs, stats, and performance.
- Promotion of source code between environments, like dev to staging or production is typically handled by release management tools like Atlassian, or Visual Studio Team Services using an automated process. Docker certificates in staging and production are typically managed by your ops team instead of developers.

Choosing a Base Image

Recall from the discussion in Chapter 2 that a Docker image is defined using a Dockerfile, which is a set of commands that are executed to create the image. Commands can include tasks such as downloading and installing applications or packages, copying files, setting environment variables, configuring what to execute when a container is started, and more. Each command in a Dockerfile is built as a separate layer and Dockerfile commands are stacked on top of one another in a hierarchy of layers. What this means is that development teams can take advantage of layering to create a base image that includes any shared components or configuration settings. Teams can start with this base image and customize as needed for improved reusability.

Choosing a base image for your microservice depends on a number of factors. The first consideration is whether you're planning to run on specialized hardware. For example, if you are building an IoT (Internet of Things) solution with Docker, one option to consider is the Resin.IO base images (https://hub.docker.com/u/resin/), which are specially tuned images that can run on devices like the Raspberry Pi or Intel Edison.

For a minimalist starting point, Alpine is a popular Linux image that is only 5 MB in size. The key benefit to Alpine is not only its small size, but that it includes a number of packages available via the Alpine Package Kit (APK) that add on key features so you can get exactly the minimum dependencies your application needs. For more information, see http://wiki.alpinelinux.org/.

If you do need full control, you don't have to use the official images. You can build your own base Dockerfile (using "FROM scratch," a reserved word for an empty image) as this provides you full control over how you're your image is created. Along with full control comes the responsibility to fully maintain and patch your Docker image.

The most common choice for base images for teams is to start with official base images from Docker Hub (http://hub.docker.com). There are a number of official images for programming languages like ASP.NET, Node, Go, Python, Java, and so on. The advantage to this approach is that someone else is responsible for the security, maintenance, performance, and versioning of the images. There is a level of trust involved here as you are trusting that the maintainers of the official image have made it secure, optimized for performance, and are regularly maintaining the image. The other benefit is that the official repository include a collection of images, grouped using tags, that include different configurations. For example, the official Java repository has tagged images that include

different Java Runtime Environment (JRE) versions, like Open JDK 6, 7, or 8.

For many of the official platform images, you'll find a number of tags that can be confusing to developers new to Docker or Linux. For example, Figure 4.1 shows some of the different tags used in the official Node image at https://hub.docker.com/_/node/.

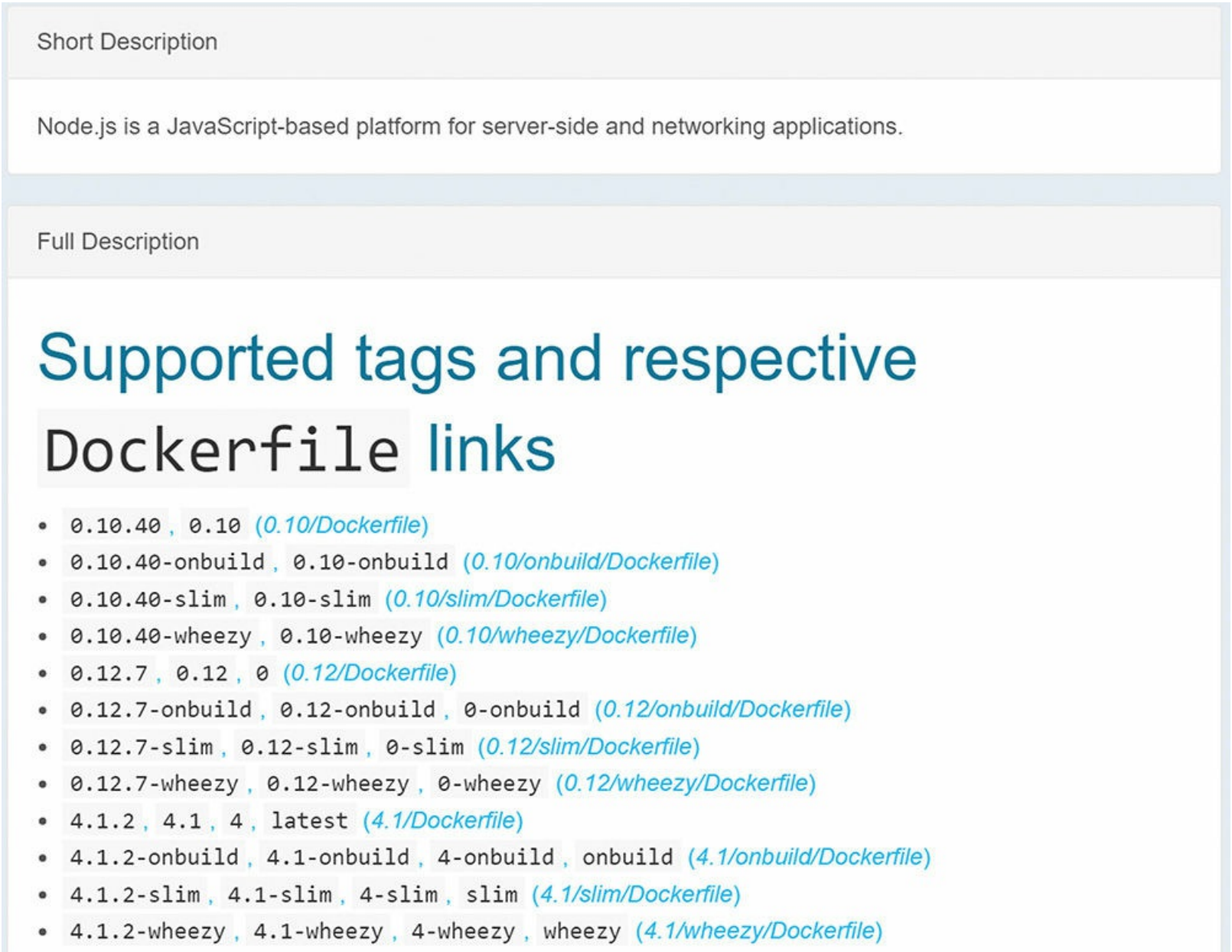

FIGURE 4.1: Image tags for the official Node Docker image

Let's review these tags, as you'll find many of them are commonly used in other Docker Hub images:

- **latest:** The latest version of the image. If you do not specify a tag when pulling an image, this is the default.
- **slim:** A minimalist image that doesn't include common packages or utilities that are included in the default image. Choose this image if the size of the image is important to your team, as the slim image is often half the size of the full image. Another reason to use the slim image is that smaller images inherently have a smaller attack surface, so they are generally considered more secure.
- **jessie/wheezy/sid/stretch:** These tags represent codenames for either versions or branches of the Debian operating system, named after characters from the Toy Story movie. **jessie** and

wheezy represent Debian OS versions 6.0 and 7.0 respectively, and **sid** is the codename for the unstable trunk, and **stretch** is the codename for the testing branch.

- **precise/trusty/vivid/wily:** These tags represent codenames for versions 12.04 LTS, 14.04 LTS, 15.04, and 15.10 of the Ubuntu operating system.
- **onbuild:** The onbuild tag represents a version of the image that includes onbuild Dockerfile commands which are designed to be used as a base image for dev and test. In normal Dockerfile commands, the execution happens when the image is created. Onbuild Dockerfile commands are different in that execution is *deferred*, and instead is executed with the downstream build.

Build a Hierarchy of Images

To get a better understanding of how to factor real-world images, we'll look at the hierarchy of the node:latest image at the time of this writing, starting with the very first image in the hierarchy

- **debian:jessie:** This Dockerfile starts from scratch, meaning it is a base image. It includes a single command to add "rootfs," the root filesystem which also adds command line tools like bash, cat, and ping. It also sets bash as the default command if you were to create a Debian container.
- **buildpack-deps:jessie-curl:** This Dockerfile starts from debian:jessie and runs the apt-get package manager to install curl, wget, and ca-certificates.
- **buildpack-deps:jessie-scm:** This Dockerfile starts from buildpack-deps:jessie-curl and runs the apt-get package manager to install source control management (SCM) tools like Git, Mercurial, and Subversion.
- **buildpack-deps:jessie:** This Dockerfile starts from buildpack-deps:jessiescm and runs the apt-get package manager to install developer utilities like GCC, Libtool, and Make.
- **node:** This Dockerfile starts from buildpack-deps:jessie and sets Node up and running by installing certificates, setting environment variables, and downloading/configuring Node. It also sets Node as the default command to run when the container starts.

As you can see from the list above, each Dockerfile is factored for reuse, so instead of putting all utilities like curl, Git, and GCC into one Dockerfile, they are in separate Dockerfiles so that developers can pick and choose which dependencies they need for their applications. To better understand how all the Dockerfiles in an image are layered, you can use the free web utility https://imagelayers.io. Figure 4.2 shows a comparison of the NodeJS **latest** (left side) and **slim** (right side) Docker images. On the left side you see how all the five Dockerfiles in the listing above are combined into separate Docker layers, starting with the first debian:jessie Dockerfile that adds the rootfs filesystem all the way to the final command to start Node in the node Dockerfile. If you look closely, the only difference between the latest and slim Dockerfiles is that the slim image saves space by skipping the source control management and developer tool Dockerfiles from the buildpack-deps:jessie-scm (122MB) and buildpack-deps:jessie (315MB) Dockerfiles.

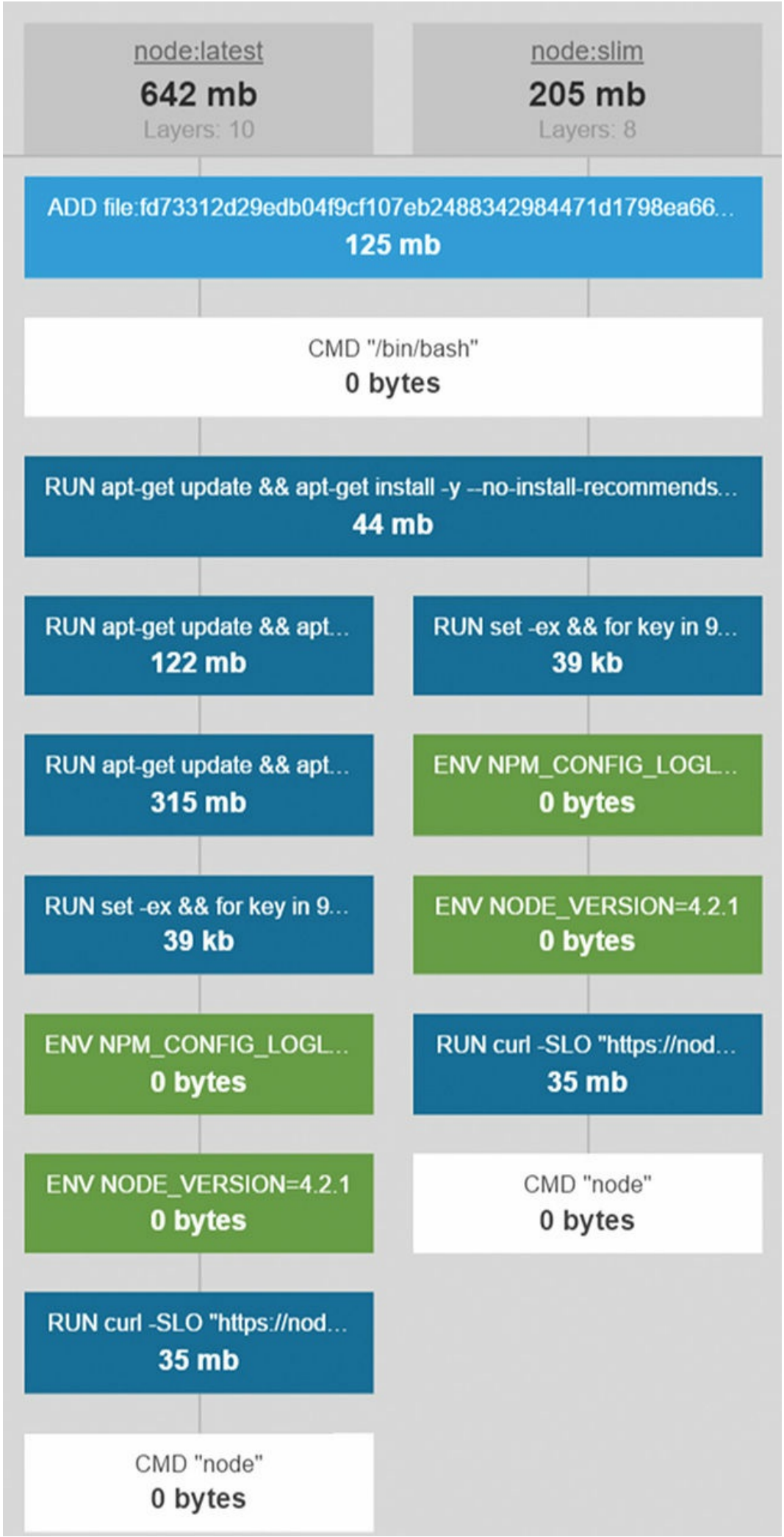
node:latest
642 mb
Layers: 10
node:slim
205 mb
Layers: 8
ADD file:fd73312d29edb04f9cf107eb2488342984471d1798ea66...
125 mb
CMD "/bin/bash"
0 bytes
RUN apt-get update && apt-get install -y --no-install-recommends...
44 mb
RUN apt-get update && apt...
122 mb
RUN set -ex && for key in 9...
39 kb
RUN apt-get update && apt...
315 mb
ENV NPM_CONFIG_LOGL...
0 bytes
RUN set -ex && for key in 9...
39 kb
ENV NODE_VERSION=4.2.1
0 bytes
ENV NPM_CONFIG_LOGL...
0 bytes
RUN curl -SLO "https://nod...
35 mb
ENV NODE_VERSION=4.2.1
0 bytes
CMD "node"
0 bytes
RUN curl -SLO "https://nod...
35 mb
CMD "node"
0 bytes

FIGURE 4.2: Comparing the latest and slim Docker images using https://imagelayers.io

For your images, you can start from an official image from Docker Hub, and then define separate Dockerfiles with any common tools or utilities your organization wants to reuse, like unit testing frameworks, or web tools like Gulp, NPM, or Bower. This enables you to build a hierarchy of images, similar to the following:

- Start with the official images from the Docker Hub.
- Build a set of team images, represented as Dockerfiles, that inherit from the official image and include common packages, files, utilities, or specific configurations that you want to reuse across microservices.
- Each microservice then has its own image and corresponding Dockerfile that inherits from the team image and includes the microservice code.

> **Official Image Dockerfiles are Open Source**
>
> All Dockerfiles that build images are open source. Simply click the Dockerfile links, like the ones from the Docker Hub pictured above, to see exactly how the image was created.

Automated Builds

To make this process easier, and to ensure only quality builds end up in your Docker Hub repository, you can automate the process of building images using continuous integration (CI) tools. For example, the official ASP.NET image uses CircleCI and GitHub hooks to automate updating the official image at http://bit.ly/aspnetci. Similarly, Docker Hub adds the capability to automate building and updating images from GitHub or Bitbucket by adding a commit hook so that when you push a commit, a Docker image build is triggered.

What happens if your base image is updated with a security fix? You can use a repository link which links your image repository to another repository, like the one for your base image. Doing this enables you to automatically rebuild your images based on changes to the underlying base image.

Organize your Images Using Tags

Like official images, when you build your images, you will want to logically organize them by tag based on the matrix of software dependencies and operating systems for your organization. For your application, another common convention, as shown in the Node official image, is to organize by base image, like jessie and slim base images, and then tags for each version (using the major/minor/patch versioning convention). The result might be something like this:

- 2.1.1-slim, 2.1-slim, 2-slim, latest
- 2.1.1-jessie, 2.1-jessie, 2-jessie, latest
- 1.0.1-slim, 1.0-slim, 1-slim
- 1.0.1-jessie, 1.0-jessie, 1-jessie

Remember to keep it simple, as even adding the two operating systems (jessie and slim) creates a large matrix for you to build and maintain. Don't over-architect dozens of Dockerfiles in a complex hierarchy if you don't actually need to! Also remember to follow Docker's best practices for creating

Dockerfiles: https://docs.docker.com/engine/articles/dockerfile_best-practices/ which help ensure you're designing your Dockerfiles to take advantage of Docker's layering system as much as possible.

Sharing Images in your Organization

Most organizations do not publish their Docker images to public repositories on Docker Hub. Instead, they opt for more private and secure options. There are a number of different options to consider for doing this.

- Docker, Inc. provides a Docker image named **registry** that you can use and run on any Docker host. In this example, you are fully responsible for managing your images and the underlying data store. As the registry is going to contain all the images your team will use, ensuring you have durable, reliable storage for those images is important. You can find instructions for how to configure the Docker registry image with Azure Blob Storage at http://bit.ly/azureregistry.
- Docker Hub supports private repositories. These repositories can only be accessed by authorized Docker Hub accounts and can be used to securely store images in the public cloud. To view private repository pricing plans, visit https://hub.docker.com/account/billing-plans/.
- For enterprises, Docker also provides **Docker Trusted Registry** (DTR), which is a private Docker registry that you own and manage, and can run on premises or in your choice of cloud provider. It adds a host of enterprise management features like integration with an enterprise's LDAP store (for example, Active Directory), user statistics, management, monitoring and commercial support. Azure includes built-in support for Docker Trusted Registry in the Azure Marketplace at http://bit.ly/azuredtr.
- **Quay.io** is a third-party public Docker registry similar to Docker Hub. It offers private repositories that you pay for on a monthly basis. For more information, visit https://quay.io.
- **Quay Enterprise** is similar to Docker Trusted Registry. It is a private instance of the Quay.io Docker Registry that you own and manage and can run on premises or in your favorite cloud provider.

Managing Images

Ongoing maintenance of images is also something to consider. Image maintenance tasks include understanding updates to software packages and updates to operating systems such as applying security patches or deploying bug fixes. For large enterprises where IT decisions are typically centralized, management of base images is handled by a central team that is responsible for maintaining the images, ensuring quality, security, and consistency across a team or an organization. In other organizations, where decision making is decentralized, each team or feature crew is responsible for their own image management.

Although Docker security tools really deserve their own chapter in a book, we quickly wanted to mention Docker Bench (https://dockerbench.com), a script created by Docker that is largely based on the Center for Internet Security's (CIS) Docker 1.6 Benchmark (https://bit.ly/ch4security) that checks for common best practices for deploying Docker containers in production.

Another common maintenance task that all organizations face is image bloat on your Docker hosts. As each image can be 500MB to 1GB, you can easily end up with hundreds of old or unused images in your shared dev or staging environments. To help fix this, you can set up a maintenance script that deletes images that reach a certain age using the `docker rmi` command.

Setting up your Local Dev Environment

Now that we've reviewed some of the considerations when setting up Docker for your teams, let's get your local development environment set up and ready to use Docker. Below is a quick list of common software to install for development.

Install Docker Tools

As we mentioned in Chapter 2, install Docker tools from http://docs.docker.com/windows/started/ (links for Linux and Mac OS X are also available on this page). Depending on what operating system you are using, different tools will be installed. However, they will be some form of the following:

- **Docker Engine:** This includes the Docker client and Docker daemon. The Docker client sends commands to the daemon and the daemon executes the commands.
- **Docker Kitematic:** An optional GUI app that, like the Docker client, is used to create and manage images and containers.
- **Docker Machine:** A command line interface designed to simplify creating Docker hosts.
- **Docker Compose:** A YAML-based tool to create or compose a group of Docker images.
- **Oracle VirtualBox:** The virtualization software that includes a preconfigured Linux ISO for local Docker development.

Docker for Windows and Mac Beta

In March 2016, Docker announced two new client tools named "Docker for Windows" and "Docker for Mac." Like Docker Toolbox, these tools install Docker and Docker Compose and enable you to setup and run Docker containers on your PC/Mac. While Docker Toolbox will remain an installation option for some time, it's likely that most development teams will switch to the new Docker tools once they officially release.

Install Developer Tools

Although developer tools are always a personal choice, Windows, Mac, and Linux developers can install Visual Studio Code, a free code-focused IDE (integrated development environment) from http://code.visualstudio.com that includes language support for Node, C#, Go, and many other programming languages. Code also includes built-in support for Dockerfile and Docker Compose colorization, syntax highlighting, and statement completion.

Install Windows Utilities

- To clone the samples used in the next section of this chapter, you'll need to install the Git command line available at https://git-scm.com/downloads.
- For Windows, install the Git Credential Manager to securely store credentials from http://bit.ly/gitcredentialmanager.
- To SSH into a Linux server from Windows, you can install the popular Putty client from http://www.putty.org.
- Developers using Windows 10 will soon be able to use Ubuntu and the Linux Bash shell directly from Windows, without the need to install command line utilities like SSH.

Install OSX Utilities

- To clone the samples used in the next section of this chapter, you'll need to install the Git command line available at https://git-scm.com/downloads. For OSX, this includes the osxkeychain mode, which uses the operating system's secure keychain to cache your credentials so there is no separate installation. OSX also includes command line support for SSH so there is no separate installation required.

Docker for Local Development

One of the key questions developers will need to consider when doing local development is how much of the production environment they want to replicate. Local development should focus on making development of your microservice fast and productive. In the following section, we'll talk about optimizing for local development, and then discuss ways to get your microservice production-ready. In Chapter 6, "DevOps and Continuous Delivery," we'll discuss further ways to validate our microservice including performance and load testing.

Local Development Settings

When you are primarily developing your microservice, there are a handful of common settings you'll want to set to debug and diagnose your application code.

- **Turn on tracing and set log levels:** Programming languages like Java, C#, Node, and Go all have the concept of tracing and logging. When you are first developing your app, you typically want to turn on verbose messages and set the appropriate log level to capture more information while debugging.
- **Fine-tune debugging and diagnostic settings using environment variables:** Each programming language provides a set of runtime settings for further control of the runtime and debugging information. For example, Go developers can use the Runtime package and set the GODEBUG variable to control garbage collection settings, Java developers can similarly control the JVM by setting the JAVA_OPTS environment variable.
- **Don't optimize your JavaScript:** For web front-end projects, it's best if you don't optimize your JavaScript. It's much easier to debug JavaScript or CSS files in their unminified and unbundled form.

Starting your Local Docker Host

After installing Docker Toolbox, run the "Docker Quickstart Terminal" app to start the local instance of Docker. You should see the following screenshot that includes the IP address used by the Docker VirtualBox image as shown in Figure 4.3.

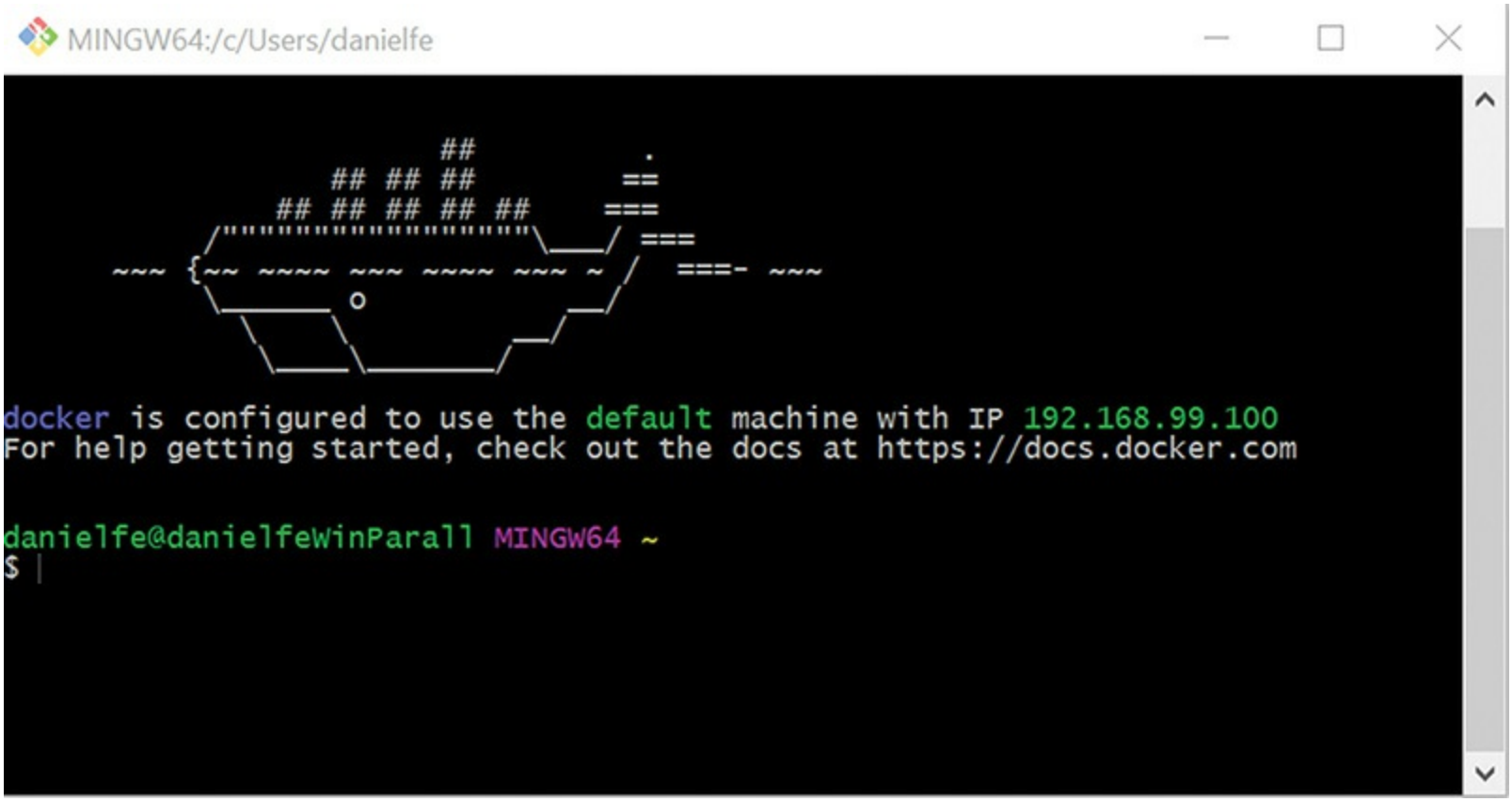

FIGURE 4.3: Starting the local Docker host

By default, the VirtualBox image will mount a file share to your user directory. For Windows, this means that "C:\Users\" is mounted as /c/Users and all your files under the Users directory are available for you to add, via the volume command, to a Docker container.

Connecting to a Docker Host

Next, we want to start a *separate* command prompt to connect to our local Docker host. To do that, you need to set the URL of the host, turn on TLS if you are using authentication, and ensure you are pointing to the right location for your Docker certificates as shown in the environment variables below. For VirtualBox, as you can see in the screenshot in Figure 4.3, it is using the *default* machine for certs and is at the *IP*: 192.168.99.100.

Set Windows Environment Variables

Click here to view code image

```
Set DOCKER_CERT_PATH=C:\Users\danielfe\.docker\machine\machines\
default
Set DOCKER_TLS_VERIFY=1
Set DOCKER_HOST=tcp://192.168.99.100:2376
```

Set OSX Environment Variables

Click here to view code image

```
export DOCKER_CERT_PATH=~/.docker/machine/machines/default
export DOCKER_TLS_VERIFY=1
export DOCKER_HOST=tcp://192.168.99.100:2376
```

The Product Catalog Microservice

If you remember from the previous chapter, the product catalog microservice is a simple ASP.NET Core 1.0 microservice that exposes a REST API to search and filter a list of products stored in an Elasticsearch backend.

Elasticsearch is a highly scalable engine that enables you to easily search and filter data. Elasticsearch is available as an official Docker image named "elasticsearch." For the ASP.NET microservice, we'll use the Nest client library to connect and query Elasticsearch.

Cloning Samples

To get started, we are going to clone the product catalog from https://Github.com/flakio/catalog to your local machine. Because later in the chapter we will use local volume mounting, it's important that you run your clone command from within a user folder like the "Documents" (C:\users\<username>\Documents) folder. This is because VirtualBox only mounts user folders by default.

From a Mac terminal, type the following:

Click here to view code image

```
cd ~
mkdir DockerBook
cd DockerBook
git clone https://github.com/flakio/catalog
```

From the Windows command prompt, type the following (replace the username as appropriate) to create a new directory named "DockerBook" and clone our repository into it as shown in Figure 4.4:

Click here to view code image

```
cd c:\Users\<username>\Documents\
mkdir DockerBook
cd DockerBook
Git clone https://github.com/flakio/catalog
```

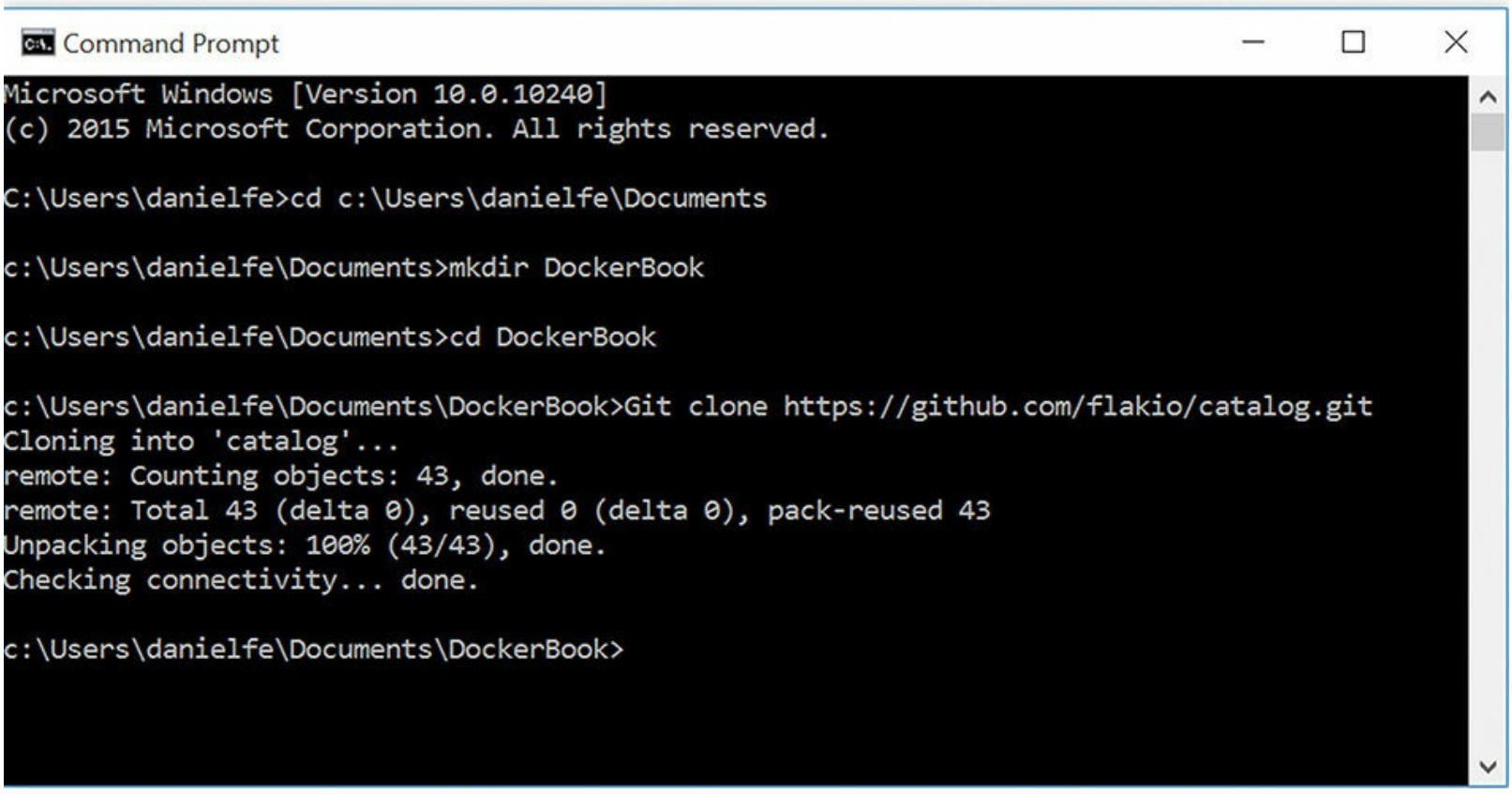

FIGURE 4.4: Cloning the Product Catalog project

Now that we have the code locally, the first thing we're going to do is set up our project using live reload.

Enabling Live Reload in a Docker Container

When we use the term "live reload," what we really mean is that we want to be able to change our source code, save, and see that change live without having to stop or restart a container.

To achieve this, there are a few things that need to happen:

- Your updated code needs to be in the container. The easiest way to do this is using Docker volume mounting.
- You need a tool to watch for code changes to restart your application if a file changes. Support for live reloading is dependent on the programming language and platform you are using. For Node, you can use NPM package nodemon, for Go, you can use the Gin utility https://github.com/codegangsta/gin to recompile when code changes, and for ASP.NET, at the time of this writing, you can use a tool named dnx-watch which watches for changes and restarts your application.
- If you add a new library using a package manager like NPM for Node, or NuGet for .NET, you need to run a restore command to ensure the new library is added into your container.

Before you get started running the sample project, you need to start the Elasticsearch backend container. You can do this with the following `docker run` command that pulls the official elasticsearch image, runs it detached (in the background), and listens on port 9200.

Click here to view code image

```
docker run -d -p 9200:9200 elasticsearch
```

You'll learn how to do this in a simpler way using linked containers and Docker Compose later in the chapter. With the data store now up, you can start the product catalog service by running:

Click here to view code image

```
docker run -d -t -p 80:80 -e "server.urls=http://*:80" -v
/c/Users/danielfe/Documents/DockerBook/ProductCatalog/Product
Catalog/src/ProductCatalog:/app thedanfernandez/productcatalog
```

> **Note**
>
> The above command is wrapped over several lines due to formatting in the printed book. You should type it as a single command on a single line.

This `docker run` command creates a detached container (-d) with tty (-t) support where port 80 of the VirtualBox VM will listen and forward a request to port 80 in the container. It uses the `-e` flag to set an environment variable to tell the ASP.NET web server to listen on port 80, mounts a volume with the `-v` flag, and bases the container on the publicly available `thedanfernandez/productcatalog` Docker image.

Volumes

The Docker volume command works just like the "-v" command line parameter we saw in Chapter 2, with the left side of the colon (:) representing the Windows/OSX/Linux directory to mount and the right side representing the destination directory inside the Docker container. Your source code from Windows/OSX/Linux is available to the VirtualBox Linux Docker host using the shared folder feature. It is then further shared from the Docker host to the container using the volume command. As shown below, VirtualBox shared folders convert "C:\" into /c/ and convert all Windows path backslashes into Linux-compatible forward slashes. For example, here are the before and after directories for our source code in the Docker host:

Windows Path

Click here to view code image

```
c:\Users\<name>\Documents\DockerBook\catalog\ProductCatalog\src\
ProductCatalog
```

Linux VirtualBox Path

Click here to view code image

```
/c/Users/<name>/Documents/DockerBook/catalog/ProductCatalog/src/
ProductCatalog
```

Finally, we use the "thedanfernandez/productcatalog" image, which is an image that when started, restores NuGet packages and starts dnx-watch to listen for changes. This image itself inherits from the "microsoft/aspnet" official image but is designed to ensure it works with the latest set of tooling by the time this book is released.

Another option for developers is to start a container interactively (giving you a standard Linux command prompt inside the running container). One key benefit to this, as mentioned in the third bullet above, is it enables you to download any new packages you might not have included in the Docker image by manually running a package restore whenever you need to. In the example below, we add the "–i" flag to start the service interactively, which will start the bash command prompt running in the container because we specified the entrypoint switch as part of the command as shown in Figure 4.5.

Click here to view code image

```
docker run -i -t -p 80:80 -e "server.urls=http://*:80" --entrypoint /
bin/bash -v
/c/Users/danielfe/Documents/DockerBook/catalog/ProductCatalog/src/
ProductCatalog:/app thedanfernandez/productcatalog
```

FIGURE 4.5: Starting a container interactively

As you can see, once you have a bash prompt inside a container, you can use standard Linux commands like the `ls` command to list the contents of the directory to ensure the volume mount command worked correctly. You will next need to run the `dnu restore` command to restore any dependencies in the project.json file into the container. Once that is complete, run the `dnx-watch` command to start listening for source code changes and to send/receive web requests.

> **ASP.NET Core 1.0 Changes**
>
> Although we've tried to make every effort possible to keep the book up-to-date with changing technology, the utilities mentioned above, `dnu` and `dnx-watch` are being rebranded to `dotnet`. What this means is that the commands will change, so for instance, `dnu` restore will become `dotnet` restore. Whereas the command names might change, the steps won't, meaning you will still need to restore `NuGet` packages and start a "`watch`" utility to watch for source code changes. The examples on `GitHub` for this book will be updated to work with any `ASP.NET Core 1.0` changes.

Once your web browser is up, you can navigate to the Docker Host IP address and the ProductsController route by going to http://ipaddress/api/products. By default, the IP address would be 192.168.99.100, which with the URL below, returns JSON product information as shown in Figure 4.6.

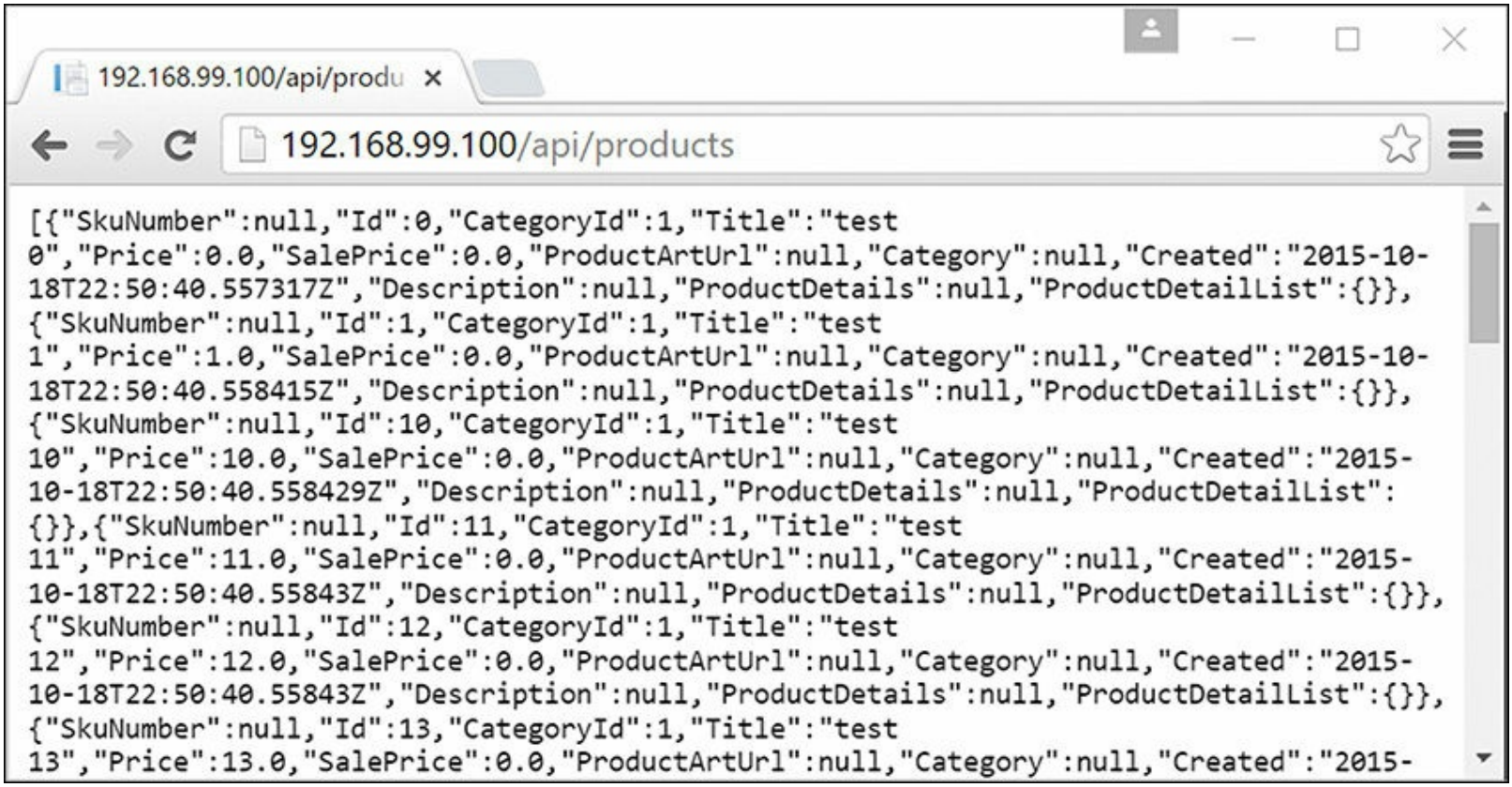

FIGURE 4.6: Calling the Products REST API

Preparing your Microservice for Production

Although we've primarily been showing off local volume mounting for development and testing purposes, most customers don't want to do local volume mounting in production, and instead want to copy all source code and dependencies directly into the container. This enables more isolation simplifies release management so that deployment is as easy as stopping and starting containers

without having to manually sync or replace source code on the host virtual machine. To prepare your microservice for production, there are a set of common steps, you'll take:

Add any Dependencies

You'll want to ensure that all required packages, tools, and dependencies that aren't already in the base image are copied or downloaded as part of the Dockerfile definition when creating the Docker image. If some of those tools or dependencies won't change every time you build, consider instead including them in a base image instead of copying them every time you build your Docker image. For our product catalog microservice, we can add NuGet package dependencies into the image by adding a RUN command to restore NuGet packages.

Optimize your Source Code

You'll also want to ensure you're optimizing your code for production. What this means is we want to do the opposite of the tasks we did for development, such as switching from debug to release, turning off debugging or changing trace settings, or removing debug-only environment variables. For ASP.NET Core 1.0, we can use the `dotnet pack` (currently dnu pack) and publish features and choose between static or dynamic compilation.

Web applications also typically have a set of client-side performance optimizations, such as minifying JavaScript, compiling CoffeeScript, or bundling and minifying LESS/SASS into CSS files. These are typically done using Grunt or Gulp task runners, and can either be done as a step before creating your Docker image or as part of the process for building your Docker image, meaning they could run inside the container itself when you execute the Docker Build command.

> **JavaScript Task Runners**
>
> Grunt (https://gruntjs.com) and Gulp (https://Gulpjs.com) are two popular client-side task runners that provide a huge ecosystem of reusable plug-ins to automate client side tasks. The tasks can be chained together in a workflow as well. For example, if you have 1,000 PNG images you need stored in AWS or Azure, you can run a task to optimize the PNGs and create responsive versions of them for different devices. You can then upload them to AWS S3 or Azure Blob storage using prebuilt plug-ins.

Although we've gotten the code ready for production, that doesn't mean the code works correctly, or is high-performing, or can handle load. We'll look at a full suite of tests for measuring code quality as part of a continuous delivery pipeline in Chapter 6.

Enable Dynamic Configuration

Your microservice should be able to run host-agnostic, meaning it can run in dev, staging, and production. You could have a number of different configuration settings like a connection string to a database, a feature flag to show or hide product recommendations, or a diagnostic setting to capture information or warnings. Each of these settings can be set when the container is created, or dynamically changed versus being hard-coded to a set value. In the Chapter 5, "Service Orchestration and Connectivity," we'll discuss different options for service discovery, like Consul or Zookeeper, that provide a more dynamic way to retreive configuration settings or connection strings.

Ensure You Test Your "Production-Ready" Microservice

Although we will cover the different types of tests in Chapter 6, and how to integrate testing into the continuous integration process, the key point is to make sure that you are testing the microservice and Docker image as much as possible to the way it will run in production.

Docker Compose

Docker Compose is a tool that enables you to define multiple containers and their dependencies in a declarative yml file. YAML (YAML Ain't Markup Language) is a declarative language similar to JSON but aimed at making it more readable by humans. Rather than surrounding everything in quotes and brackets like JSON does, it uses indentation for hierarchy. Developers like Docker Compose because it's more readable than the Docker command line and makes it easy to compose and connect multiple containers in a declarative way. The best way to understand Compose is by opening the Docker Compose file at:

Click here to view code image

```
c:\Users\<username>\Documents\DockerBook\catalog\src\ProductCatalog\
docker-compose.yml
```

or on a Mac

Click here to view code image

```
~/DockerBook/catalog/ProductCatalog/src/ProductCatalog/docker-
compose.yml.
```

> **Visual Studio Code**
>
> Visual Studio Code has awesome support for Docker Compose files, including IntelliSense, colorization, formatting, inline help (mouse over a Compose element to see help pulled directly from Docker's web site), and even real-time image name lookup (start typing the name of an image and it'll pull the image metadata directly from Docker Hub).

As you can see, this compose file does many of the same things we did on the Docker command line. Read through the file and we'll review in more detail below:

Click here to view code image

```
productcatalog:
  image: "thedanfernandez/productcatalog"
  ports:
    - "80:80"
  tty: true
  links:
    - elasticsearch
  environment:
    - server.urls=http://*:80
elasticsearch:
  image: "elasticsearch"
  ports:
    - "9200:9200"
```

At a high level, this compose file creates two containers using the arbitrary labels,

"productcatalog" and "elasticsearch." The elasticsearch container is simple to explain as it does what our previous `Docker run` instruction did: it uses the official elasticsearch image and listens on port 9200.

The "productcatalog" container represents the ASP.NET Core 1.0 microservice and has many similarities to the `Docker run` command we used before. It uses a base image from thedanfernandez/productcatalog which includes the source code "baked" into the image using the Dockerfile ADD command, which adds the source code into the image. It also does the common steps you've seen before such as opening port 80 for the host and container, turning on tty, and setting the `server.urls` environment variable to tell the `ASP.NET` web server to listen on port 80.

Linked Containers

Container linking is a special Docker feature that creates a connection between two running containers. `productcatalog` has a `link` defined to the `elasticsearch` container. Instead of having to define a brittle connection string in our app to a specific IP address to the elasticsearch data store, container linking will inject a set of environment variables into the `productcatalog` container that includes a connection string to the elasticsearch service so that two containers can easily communicate to each other without a hardcoded connection string.

One thing to keep in mind is that you cannot link containers across multiple Docker hosts as both containers must run on the same host. To achieve that, you can use Docker's overlay networking feature which provides a way to connect multiple Docker hosts using a discovery service like Consul. We will discuss Docker networking in more detail in Chapter 5.

Container Dependencies

Container linking also defines a dependency and order of instantiation for containers. Because the `productcatalog` container is linked to and depends on the `elasticsearch` container to run, Docker Compose will create the `elasticsearch` container first, and then create the `productcatalog` container.

ASP.NET and Environment Variables

ASP.NET Core 1.0 includes a fully extensible way to store configuration settings. In the example below, we have two sets of configuration settings, a JSON file named `config.json` and environment variables. Because environment variables are added after the JSON file in the `ConfigurationBuilder` class, they take precedence, and overwrite any variable with the same name defined in the `config.json` file. This is perfect for our container linking example as we can take the injected environment variables from the linked container (`ELASTICSEARCH_PORT`) and replace the connection URL to `elasticsearch` defined in `config.json` dynamically.

Click here to view code image

```
Public Iconfiguration Configuration { get; private set; }
public Startup(IapplicationEnvironment env)
{
  var builder = new ConfigurationBuilder(env.ApplicationBasePath)
    .AddJsonFile("config.json")
    .AddEnvironmentVariables();
  Configuration = builder.Build();
}
```

To run our Docker Compose file and have the containers run detached (in the background), type the following command. Your working directory needs to be the directory that contains the docker-compose.yml file. This is shown in Figure 4.7:

```
docker-compose -d up
```

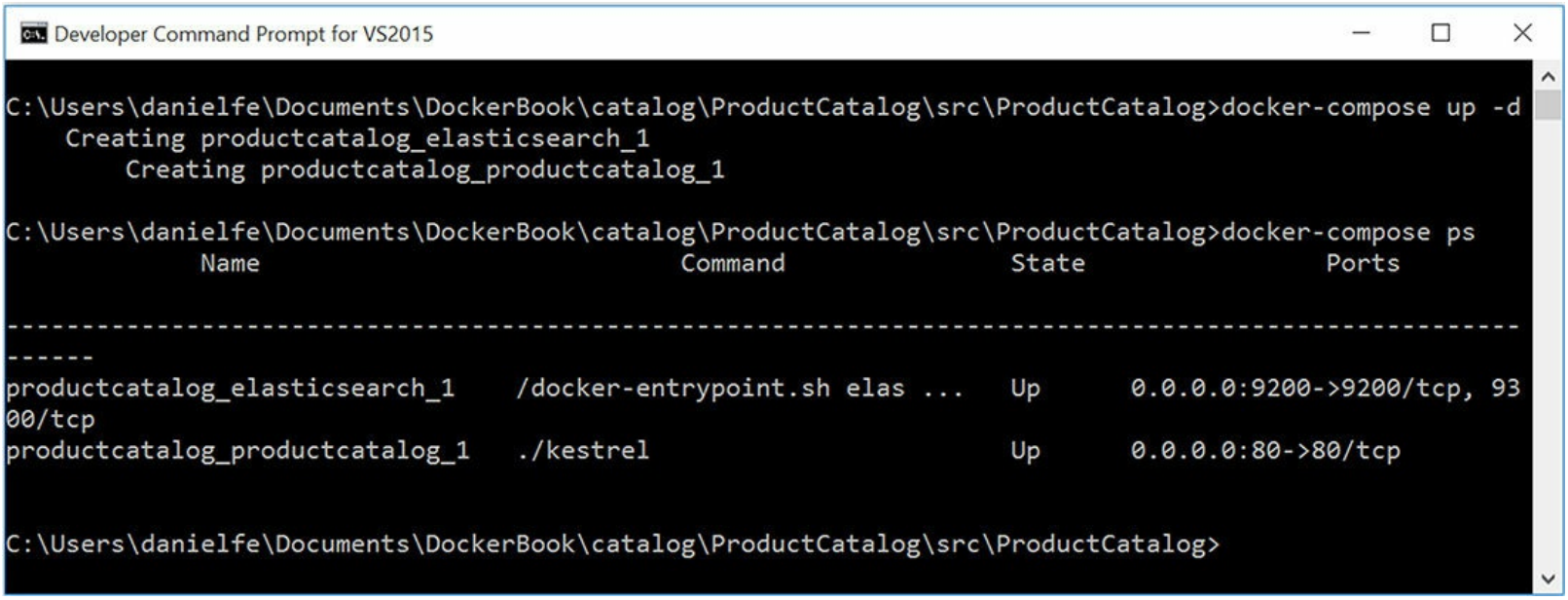

FIGURE 4.7: Running Docker Compose

Notice that Docker Compose created the `elasticsearch` container first, and then the `productcatalog` container based on the linked container dependency. The name of the running container is the lowercased folder name, so creating the elasticsearch image in the "ProductCatalog" folder created "productcatalog_elasticsearch_1." You can run further commands with Docker Compose, like `docker-compose ps` to see the status of the two containers. Similarly, we can type `docker-compose stop` to stop both containers.

Smart Restart

Docker Compose is also smart in that, where possible, it will detect the current state of containers and only change/restart if there is something newer. For example, if you change the Catalog microservice and don't change the Elasticsearch container, and then run a `docker-compose up` command, Docker Compose won't stop and recreate the Elasticsearch container because it can detect there are no changes. This is shown in Figure 4.8.

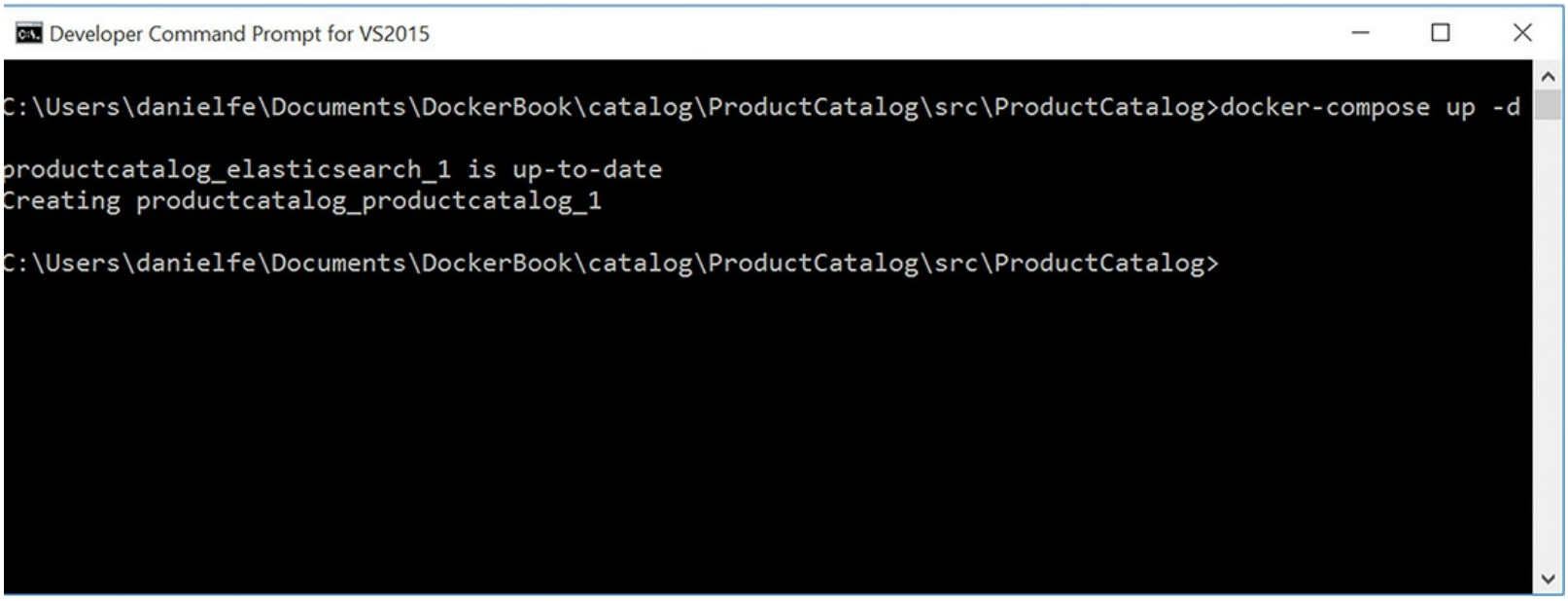

FIGURE 4.8: Docker Compose recreating only changed containers

Debugging Docker Issues

For developers new to Docker, it can be difficult to wrap your head around how to debug issues when building Docker images or trying to run containers. Below are some of the common techniques to help when you get stuck.

Unable to Connect to the Docker Host

Sometimes the hardest part is just connecting! If you can't even connect to the Docker host, there are a few steps to try. To rule out issues where the host might not be up, the Docker daemon might not be started, or port traffic might be blocked, first attempt to SSH into the virtual machine. If you can successfully run `docker ps` from an SSH client, you know the issue isn't the Docker daemon.

Another common connectivity issue is port mapping. If your virtual machine is behind some form of load balancer, you might need to ensure both that the Docker port is opened (default of 2376) and that any application-specific ports, like port 80 for a Web server, are open.

Another common issue is Docker certificates. Ensure that you are using the correct Docker certificates by verifying the DOCKER_CERT_PATH as mentioned earlier in the chapter.

Containers That Won't Start

The first thing to do is to check the Docker logs. First, get the ID for the container by using the "–a" flag to list all containers, including stopped containers as shown:

```
docker ps –a
```

Once you have the image ID, view the logs by typing

```
docker logs <id>
```

If the answer isn't immediately obvious, you can connect to the container directly as we discussed earlier in the chapter by overriding the container's entrypoint in the Docker run command as shown below:

Click here to view code image

```
docker run –t -i -p 80:80 --entrypoint=/bin/bash thedanfernandez/
productcatalog
```

This snippet replaces the "–d" (detached) flag with the "–i" flag to run interactively and overrides the Dockerfile entrypoint to instead start the bash command prompt. Once it runs, you should see a bash command prompt that you can use to further diagnose issues such as:

- ensuring all application dependencies are in the base image.
- ensuring any apps or dependencies (ex: NPM) are installed.
- ensuring any environment variables your app needs are correctly configured.

When you type "`exit`" to exit the interactive shell, the container will automatically stop, but you can go back and use the Docker logs command to view all the commands you typed into the bash shell and the corresponding command line output.

Diagnosing a Running Container

To do the same kind of diagnostics on a running container, you can either:

- attach to the container using the docker attach command.
- use the docker exec command to execute commands in the container and return the result to the command prompt using docker exec <container-id> command.

For example, you can call the `exec` command on the elasticsearch container using the first four characters of the container id, "bbd8," to view the contents of a file using the Linux cat command.

Click here to view code image

```
docker exec bbd8 cat docker-entrypoint.sh
```

This will execute the command in the container and return the output, in this case the contents of the docker-entrypoint.sh file as shown in Figure 4.9.

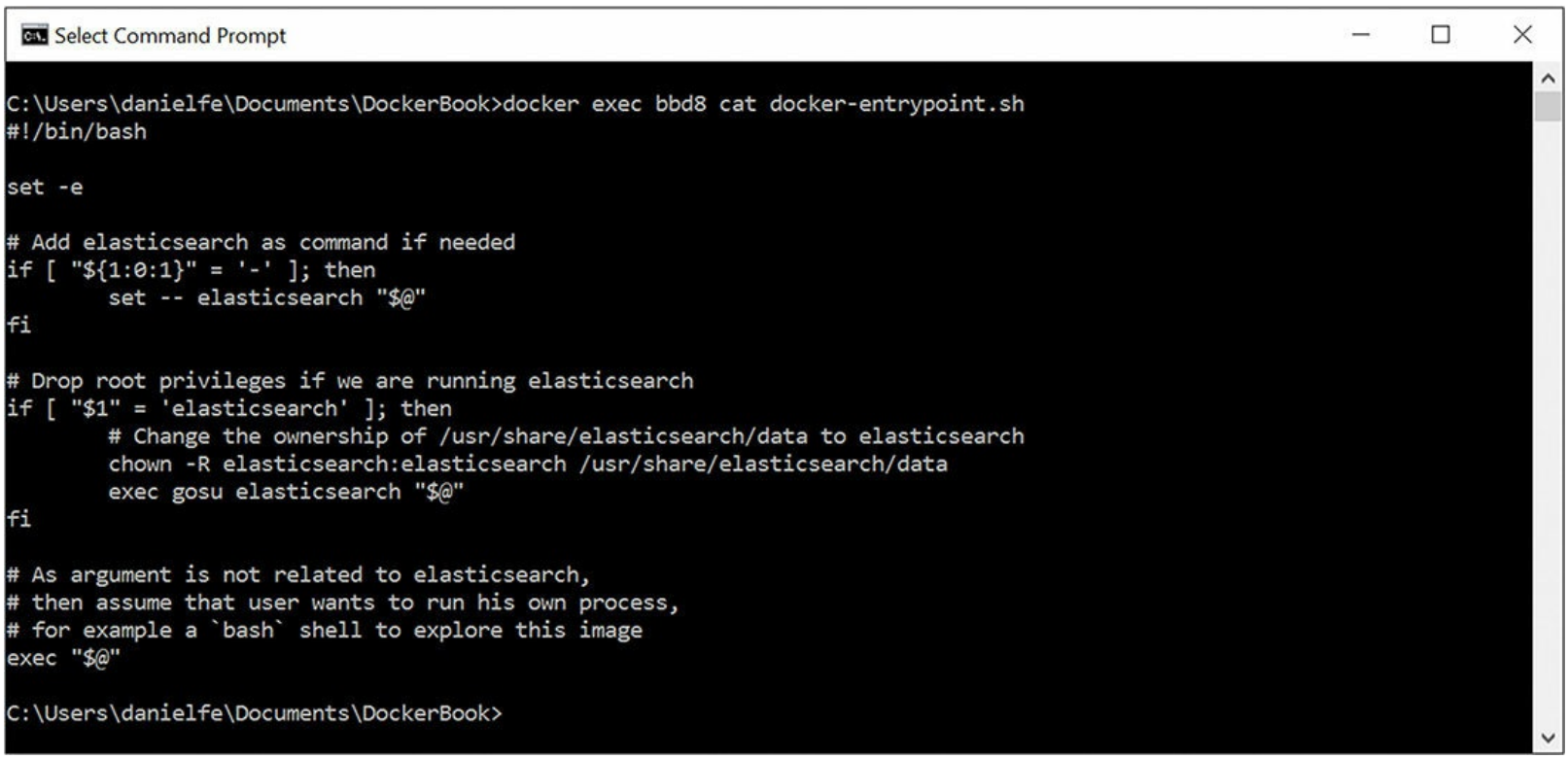

FIGURE 4.9: Using Docker Exec to see the contents of a file in a running container

Summary

In this chapter, we covered a lot of steps for how to set up local development with Docker, image management, Docker registry options, local versus production-ready development, Docker Compose, and how to diagnose issues when creating Docker images and containers.

5. Service Orchestration and Connectivity

It's likely our microservices-based application is composed of more than one service, and multiple instances of each service. So how do we deploy and manage multiple services, multiple instances of each service, and the connectivity between the services? With a microservices architecture, it's even more important that we automate everything. Orchestration tools are used to deploy and manage multiple sets of containers in production, and service discovery tools are used to help route traffic to the appropriate services.

The services used to compose our application could be deployed into a growing number of cloud-hosted compute offerings available today. Using some combination of Infrastructure as a Service (IaaS) or Platform as a Service (PaaS), the microservices can be deployed and run in their own dedicated machines or a set of shared machines. In this chapter we will discuss using a cluster of machines to host our Dockerized microservices.

Before jumping into the details of orchestration, scheduling, cluster management, and service discovery, let's start with a conceptual overview of a typical cluster used to host a microservice-based application. Figure 5.1 shows a conceptual diagram of a typical environment. In addition to the machines hosting our services, we have management and orchestration systems used to deploy and manage the services that compose our application.

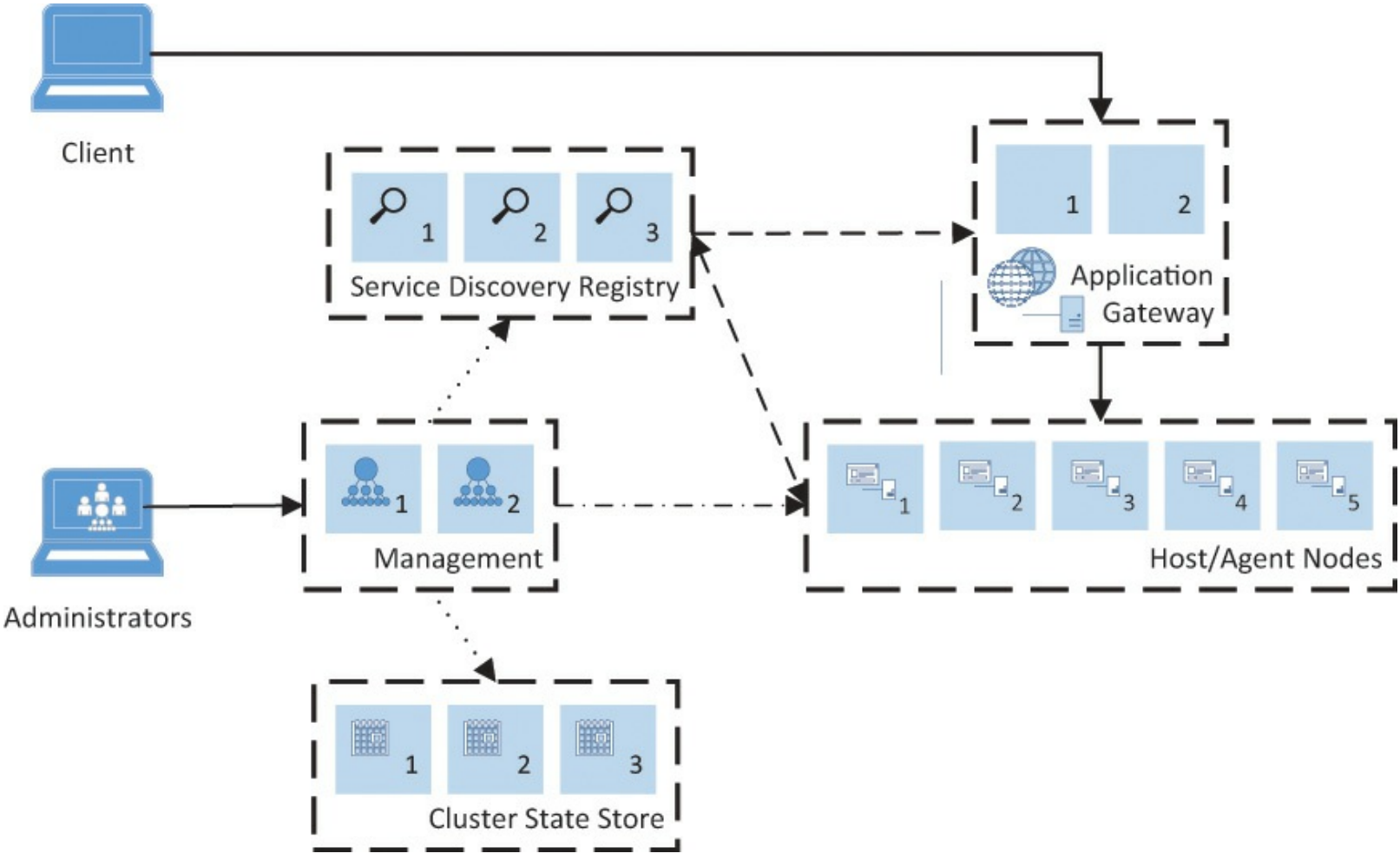

FIGURE 5.1: Conceptual overview of a cluster host environment

Conceptually the architecture is fairly simple, but the complexities are often in the details.

- *Host Nodes* are the machines our services are deployed to and run on. This is our pool of

resources doing all the heavy lifting and delivering the application functionality. Clustering and scheduling technologies will often refer to these as agents, minions, or nodes.

- *Management Services* are responsible for selecting and deploying the services to the host nodes, and often provide the API or UI. Many technologies refer to these as "masters." Technologies like Mesos with Marathon, Docker Swarm, and Kubernetes are commonly used.
- *Cluster State Store* is used by the management services to elect a leader and store cluster state, like the nodes and information about each node. Technologies like Zookeeper, Consul, and etcd are often used.
- A *Service Discovery* store is used by our services to announce that they are ready to receive requests and the endpoint they can be reached on. Consumers use this information to route requests to the service endpoints. Zookeeper, Consul, and etcd are popular options for storing service discovery information.
- The *Application Gateway* is used for routing and load balancing inbound traffic into the cluster. In addition to routing and load balancing, this service can also perform tasks like SSL offload, request aggregation, and authentication. NGINX and HAProxy are popular technologies.
- *Artifact Store/Image Registry* is not shown in the diagram, the artifact store is used to manage deployment of artifacts like the various files used when scheduling services in the cluster. Docker images are artifacts that are commonly stored in a public hosted image registry such as Docker Hub. Other hosted options are available and there are some very good reasons to host these services yourself, including security, performance, and availability.

Some of these services are often deployed to the same machines. For example, the management services can actually be running on the same machines hosting the cluster state store. Sometimes the cluster state store and service discovery registry share a common deployment of a KV store. The application gateway can be running on the same cluster host nodes.

In this chapter we will learn more about these concepts and the technologies used in creating an environment to deploy and run our microservices-based applications.

Orchestration

In the context of infrastructure and systems management, orchestration is a fairly general term that is often used to refer to cluster management, scheduling of compute tasks, and the provisioning and de-provisioning of host machines. It includes automating resource allocation and distribution, with the goal of optimizing the process of building, deploying, and destroying computing tasks. In this case, the tasks we're referring to are microservices, such as those deployed in Docker containers.

In this context orchestration consists of provisioning nodes, cluster management, and container scheduling.

Provisioning is the process of bringing new nodes online and getting them ready to perform work. In addition to creating the virtual machine, this often involves initializing the node with cluster management software and adding it into a cluster. Provisioning would also commonly include resources other than compute, like networking, data storage services, monitoring services, and other cloud provider services. Provisioning can be the result of manual administration, or automated scaling implementation used to increase the cluster pool size.

Cluster management involves sending tasks to nodes, adding and removing nodes, and managing

active processes. Typically there is at least one machine that acts as cluster manager. This machine(s) is responsible for delegating tasks, identifying failures, and synchronizing changes to the state of the application. Cluster management is very closely related to scheduling, and in many cases the same tool is used for both.

Scheduling is the process of running specific application tasks and services against specific nodes in the cluster. Scheduling defines how a service should be executed. Schedulers are responsible for comparing the service definition with the resources available in the cluster and determining the best way to run the service. Schedulers integrate closely with cluster managers because schedulers need to be aware of each host and its available resources. For a more detailed explanation, see the section on Scheduling.

> **Service Connectivity**
>
> Although service discovery, application gateways, and network overlays are not necessarily a part of orchestration, we will cover them in this chapter as they are closely related to setting up a cluster and running services in a cluster.

Service Discovery and Application Gateways are used to route traffic to the service instances deployed in the cluster. We typically have multiple instances of our service running on nodes within the cluster, as determined by the scheduler, and we need a way to discover their location and route traffic to them. Some container orchestration tools will provide this functionality.

Overlay networks enable us to create a single network that sits on top of multiple networks beneath. We can deploy containers and services to the overlay network, and they can communicate with other containers on the same overlay network without having to worry about the complexities of the networks beneath.

Let's start with provisioning and bootstrapping the virtual machine with the necessary cluster management and orchestration tools.

Provisioning

We are going to need some machines to run our application, and they will need to be initialized with cluster management software. In addition to the virtual machines needed to run our application, we will need to provision storage accounts, availability sets, virtual networks, and load balancers. We might even need to provision some additional Microsoft Azure hosted services like a Service Bus, Azure DocumentDB, or Azure SQL Database.

We can break provisioning down into two high level concerns: outside the box and inside the box. Provisioning outside the box would include creating all the correct resources like networking, virtual machines, and similar. Inside the box would be anything that needs to set up inside the service, such as cluster management software.

A big challenge with infrastructure has been the capability to reliably recreate an environment. Machines and networks are provisioned, configurations are changed, and other machines are added or removed. Even if all this is documented, tracking the changes and working through manual steps is error-prone, and rarely ends with exactly the same environment setup. Automating infrastructure provisioning can help here, and we can apply a lot of the best practices for managing application code to managing our infrastructure.

Let's start with a quick overview of infrastructure as code and the services used in Microsoft Azure to provision and manage our resources.

Infrastructure as Code

Infrastructure as code, also known as programmable infrastructure, is a way of building and managing an application's entire environment through predefined code. It enables developers and administrators to recreate the same environment on multiple physical machines. An environment that can be reproduced is crucial to efficient development, testing, and deployment. As the size and complexity of a service increases, so does the difficulty in reproducing it consistently.

Administrators often rely on scripts to automate this process. Infrastructure as code performs a similar service while also adding automation, version control, true cross-platform support, and other features. Infrastructure as code typically consists of one or more *definition files* written in a high-level language, which contain instructions that are read and interpreted by a separate tool.

Challenges with Manual Deployments

Infrastructure as code seeks to solve the problem of managing and deploying infrastructure. Existing approaches that often involve manual intervention can make it difficult to keep track of how environments have been configured and how they are currently being managed. For instance, it can be possible to create an environment for a very small application following manual steps in a document, but as the application and infrastructure grows in size and complexity, the chance for human error does as well.

Applications of all sizes can benefit from automation. With larger applications we can quickly find ourselves in situations where multiple machines or environments are using inconsistent configurations for the same application. Tracking changes across the environment and who made them can be nearly impossible. Deploying multiple environments, such as dev, test, and prod in a consistent way becomes error-prone and challenging. It can be impossible to audit and determine who made changes to the environment, and this can impact potential compliance requirements. We can avoid much of that pain through automation.

Infrastructure as Code as a Solution

With infrastructure as code, we can define environments that are easily updatable, consistently reproducible, distributable, version-controlled, and disposable.

Updating the Environment

In an agile development project, application requirements can change rapidly. An environment built for last week's version might not work for today's version. Instead of having to manually tear down and rebuild your test environment from scratch, you can simply change how your environment is defined and automatically build a new environment tailored to the latest version of the application.

Pets vs. Cattle

The pets and cattle metaphor is commonly used to describe how we should think about servers today—treat them like cattle. They are a part of a herd, almost identical, and when they get sick we replace them with another one instead of nursing them back to

health. If any server in the organization is known by name and pain is felt when it's down, then it's being treated like a pet.

Maintaining Consistency

Consistency is key when it comes to microservices. When it's time to deploy, having a way of accurately reproducing an environment on another system greatly reduces the chance of an error. Because infrastructure as code relies on predefined code to create the environment, each instance of the environment is guaranteed to run the same. This also makes the environment easy to share with developers, quality assurance, or operations.

Tracking Changes

Because infrastructure as code relies on definitions instead of actual environments, changes can easily be tracked using a version control system such as Git, Subversion, or Mercurial. Not only does this help keep track of changes to the environment, but it also helps administrators maintain different versions of the environment for different versions of the application.

Removing Old Environments

If an environment is no longer needed, traditionally the programs and services belonging to that environment had to be removed manually. With infrastructure as code, you can define tasks that automate this process. For instance, in Microsoft Azure, you can remove an entire environment by simply deleting resource group. By using a compartmentalized service such as Docker, you can even remove a resource without affecting the host machine.

Test-Driven Infrastructure

Infrastructure as code changes the way we need to look at testing. One approach to doing this is through a test-driven infrastructure. Similar to test-driven development, a test-driven infrastructure is designed around a predetermined set of test cases that must pass successfully.

Azure Resource Manager

The Azure Resource Manager (ARM) can be used in an infrastructure as code approach to resource provisioning and management in Azure. ARM enables us to define the desired state of our infrastructure using a JSON-based templating language. All the intermediate steps—provisioning, redundancy, dependency management, and fault tolerance—are handled seamlessly by Azure. ARM templates are completely self-contained and can easily be deployed to and from Azure. The Azure Resource Manager must carry out the tasks of creating or modifying resources, as defined in an ARM template. A resource is just a virtual machine, virtual network, storage account, load balancer, or similar. Azure Resource Manager also enables us to organize resources into groups, and organize resources through tagging.

Benefits:

- Deploy, manage, and monitor all resources in a group.
- Repeatable deployments.

- Declarative templates for defining deployments.
- Define resource dependencies and ARM will optimize deployments.
- Apply role-based access control to resources in the group.
- Logically organize resource by tags.

ARM is flexible and enables us to use either an imperative or declarative approach. That means we can define our infrastructure in a template and then send that to ARM to handle the provisioning, or we can create a script to call APIs for provisioning as we might have done in the past. Using the declarative approach, templates can be created and managed in source control, where changes to the infrastructure can be versioned and tracked. We can make changes to the template and pass it to ARM to apply those changes to an existing resource group, or create a completely new one. We can rest easy knowing that we can quickly and reliably recreate our infrastructure.

ARM Templating Language

The ARM templating language is a JSON-based document for defining an Azure infrastructure. A template contains configuration details and metadata about each resource, including the type of resource and where it should be deployed. In addition to schema and content version, a template contains four top level elements.

Top level elements:

- **Parameters** provide input for when a deployment is executed and can be used to customize resource deployment. An example of this is the number of machines we want to deploy in the cluster.
- **Variables** are used to simplify template language expressions. One use for variables might be a string we commonly use through the template, or formatting of a parameter to be used throughput the template.
- **Resources** are the types of services to be deployed or updated in the resource group. Examples of resource include virtual machines, virtual networks, and storage accounts.
- **Outputs** are values that are returned after a deployment. These can be used to provide DNS names or IP addresses created as part of the deployment.

The following JSON document shows the basic structure of a template. From this simple document we only need to define a resource in the ‘resources’ section.

Click here to view code image

```
{
   "$schema": "http://schema.management.azure.com/schemas/2015-01-01/
deploymentTemplate.json#",
   "contentVersion": "",
   "parameters": {  },
   "variables": {  },
   "resources": [  ],
   "outputs": {  }
}
```

Let’s have a closer look at each section. Before we do that we need to first cover template functions and expressions.

Functions

Expressions and functions extend the JSON available in the template. This enables us to create values that are not strictly literal. Expressions are enclosed in square brackets "[]," and they are evaluated when the template is deployed. Template functions can be used for things like referencing parameters, variables, and value manipulation such as type conversions, addition, math calculations, and string concatenation.

As in JavaScript, function calls are formatted as functionName(arg1, arg2, arg3)

Click here to view code image

```
"variables": {
  "location": "[resourceGroup().location]",
"masterDNSPrefix":"[concat(parameters('dnsNamePrefix'),'mgmt')]",
"agentCount": "[parameters('agentCount')]"
}
```

Parameters

The parameters section is used to define values that can be set when deploying the resources. We can then use the parameter values throughout the template to set values for the deployed resources.

The JSON example following defines some parameters starting with the "agentCount," followed by "masterCount." For each parameter we must define a type, and we can optionally define default values and some other constraints, as well as some metadata that can contain additional information for other consuming the template. As we see in the following, we have set a default value to the "agentCount" parameter and we have declared the minimum value for this parameter of 1 and a maximum value of 40.

Click here to view code image

```
"parameters": {
  "agentCount": {
  "type": "int",
  "defaultValue": 1,
  "metadata": {
    "description": "The number of Mesos agents for the cluster."
  },
  "minValue":1,
  "maxValue":40
  },
"masterCount": {
  "type": "int",
  "defaultValue": 1,
  "allowedValues": [1, 3, 5],
  "metadata": {
    "description": "The number of Mesos masters for the cluster."
  }
}
```

Variables

In the variables section we can construct values that can be used to simplify template language expressions. These variables are commonly based on values from the parameters as we saw with "agentCount."

The values can be transformed using ARM template language expressions like the "concat" function. We can see this with the "masterDNSPrefix" variable in the following example. The value for "masterDNSPrefix" will contain a string value of whatever was passed in the parameter "dnsNamePrefix" with "mgmt" concatenated at the end of it. Now we can simply reference this variable in the template instead of having to perform this concatenation. This improves the readability and maintenance of the template.

Click here to view code image

```
"variables": {
  "masterDNSPrefix":"[concat(parameters('dnsNamePrefix'),'mgmt')]",
  "agentDNSNamePrefix":"[concat(parameters('dnsNamePrefix'),'agents')]"
  "agentCount": "[parameters('agentCount')]",
  "masterCount": "[parameters('masterCount')]",
  "agentVMSize": "[parameters('agentVMSize')]",
  "sshRSAPublicKey": "[parameters('sshRSAPublicKey')]",
  "adminUsername": "azureuser"
}
```

Resources

In the resources section we define the resources that are deployed and updated. This section is the bulk of most templates and contains a list of resources. The following template snippet defines an Azure Container Service (ACS) and some parameters. The "apiVersion," "type," and "name" elements are required elements for every resource. The apiVersion must match a supported version for the "type" of resource defined, and the name defines a unique name for the resource that can be used to reference it. There are some additional documented elements not shown, like the "dependsOn" element used to define resource dependencies that need to be provisioned first. The "properties" element contains all the resource properties. For example, the "orchestratorProfile" is a property of the container service as we have defined it here.

Click here to view code image

```
"resources": [
  {
    "apiVersion": "2015-11-01-preview",
    "type": "Microsoft.ContainerService/containerServices",
    "location": "[resourceGroup().location]",
    "name":"[concat('containerservice-',resourceGroup().name)]",
    "properties": {
      "orchestratorProfile": {
        "orchestratorType": "Swarm"
      }
    }
  }
]
```

Outputs

In the optional outputs section we can specify values that are returned from deployment. For example, we could return the Fully Qualified Domain Name (FQDN) for the master and agent nodes created in the deployment. This could then be used by a user or a script to connect to these endpoints using this domain name.

Click here to view code image

```
"outputs": {
  "masterFQDN": {
    "type": "string",
    "value":
"[reference(concat('Microsoft.ContainerService/containerServices/',
'containerservice-', resourceGroup().name)).masterProfile.fqdn]"
  },
  "agentFQDN": {
    "type": "string",
    "value": "[reference(concat('Microsoft.ContainerService/
containerServices/', 'containerservice-', resourceGroup().name)).
agentPoolProfiles[0].fqdn]"
  }
}
```

Using an ARM template, we can repeatedly deploy our infrastructure through the entire lifecycle and have confidence that the resources are deployed in a consistent manner. We define our infrastructure in a template that is managed in source control, where we can track changes to the infrastructure. When the template is changed we can apply the new template to existing deployments and the Azure Resource Manager will determine what needs to change and make the necessary changes based on the current state of the template and the existing state of the deployment.

> **Linked Templates**
>
> As templates grow, we can compose templates from other templates through linking. This is a great technique used in managing templates, especially as they grow in size and complexity. We could create a template responsible for deploying Consul, for example, and then reference that template in our Docker Swarm template.

Parameters for each environment can be maintained in source control or some other configuration store. Like environment configuration for an application, we would generally store this in the operations source control repository or a special database. Parameters would generally be things like the names, locations, and number of instances of the various resources. We might want to deploy a cluster into a staging or test environment that has a fewer number of nodes or smaller nodes. Instead of making the number of nodes in the cluster part of the template, we can make it a parameter. Then, when we are deploying a template we can pass the parameter value containing the number of nodes in the deployment suitable for the environment. The deployments of our cluster are consistent and the same across environments, with some variation that can be controlled with parameters and tested, like the number of nodes in this case.

Inside the Box

Once the Azure Resource Manager (ARM) has provisioned the virtual machines and other services outside the box, there is often a need for some provisioning inside the box. Often we will need to install some software, like our cluster management software. Once the virtual machine is ready, we can have ARM invoke a script and pass some context in to perform the provisioning inside the box. Some ARM extensions are available and can be defined in the template, with the context passed, such as the information needed to join the cluster. A couple of extensions that are commonly used when provisioning machines are the script extension and the Docker extension. The script extension enables

us to define a set of scripts to download to the machine and a command to execute. The Docker extension will ensure the Docker daemon is installed on the machines, and use a JSON document matching the Docker Compose format to start Docker containers on the instance after they have been provisioned.

These extensions are commonly used to bootstrap the provisioned machines with cluster management software and join the cluster. For example, we can use the script extension to install the Apache Mesos agent on each node defined in the template. We could use the Docker extension to install the Docker Swarm daemon and register nodes with the cluster manager.

In addition to using extensions to bootstrap our machine, we can embed a cloud-config into the template as custom data. Cloud-config files are scripts designed to be run by the cloud-init process, which is used for initial configuration on the first boot of a server. We can install our cluster management software through this mechanism as well. The cloud-config works like a configuration file for common tasks and enables you to define scripts for more complex functionality. More information on cloud-init is available in the documentation at https://cloudinit.readthedocs.org/en/latest/. Not all virtual machine images in Azure support this, or have cloud-init enabled.

Azure Container Service

Microsoft Azure Container Service (ACS) is an Azure Resource Provider for creating and managing a cluster of machines that act as container hosts. ACS makes it extremely easy to provision a production-ready cluster using popular open source cluster management and scheduling tools.

A single resource provider can be used to provision a Mesos or Docker Swarm cluster, which are some technologies we will cover later in this chapter, in the scheduling section. The ACS resource provider requires only a handful of properties to deploy an entire cluster. The properties can even be defined at deployment using ARM template parameters and variables discussed earlier in this chapter.

ACS Resource Provider

The following ARM resource snippet could be used to provision a cluster in Azure using a configurable set of parameters. Note the resource type defined is "Microsoft.ContainerService/containerService." A location is provided which will place it in the same location in which the resource group was created, and the name of this resource will simply prepend "containerservice-" to the resource group name. The more interesting parts are the properties of the resource provider, where the values for the properties are being set using ARM variables.

ACS Resource Provider Properties:

- **orchestrationProfile** element is used to define the type of orchestration tooling to deploy.
- **masterProfile** element is where we can define properties for the master nodes, such as the number of nodes to create, and a prefix used for creating DNS entries for the master node load balancer.
- **agentPoolProfiles** element enables us to define multiple collections of agent nodes, each having a different name, count, size, and dnsPrefix.
- **linuxProfile** element is where we can define system property settings for all the nodes provisioned, like the credentials.

Click here to view code image

```
{
  "apiVersion": "2015-11-01-preview",
  "type": "Microsoft.ContainerService/containerServices",
  "location": "[resourceGroup().location]",
  "name":"[concat('containerservice-',resourceGroup().name)]",
  "properties": {
    "orchestratorProfile": {
      "orchestratorType": "Mesos"
    },
    "masterProfile": {
      "count": "[variables('masterCount')]",
      "dnsPrefix": "[variables('mastersEndpointDNSNamePrefix')]"
    },
    "agentPoolProfiles": [
      {
        "name": "agentpools",
        "count": "[variables('agentCount')]",
        "vmSize": "[variables('agentVMSize')]",
        "dnsPrefix": "[variables('agentsEndpointDNSNamePrefix')]"
       }
     ],
     "linuxProfile": {
      "adminUsername": "[variables('adminUsername')]",
      "ssh": {
        "publicKeys": [
          {
            "keyData": "[variables('sshRSAPublicKey')]"
          }
        ]
      }
    }
  }
}
```

As we can see here, Microsoft Azure makes it very easy to provision production-ready Mesos and Docker Swarm clusters.

The flak.io sample application includes detailed steps for provisioning a new cluster and deploying the flak.io ecommerce sample into the cluster at http://flak.io.

Deploying an ARM Template to Azure

Azure supports several methods for deploying templates, including PowerShell, the Azure Command Line Interface (CLI), or a standard REST API. PowerShell and Azure CLI require you to log into your Azure account and select the subscription that you want to deploy to. You will also need to create a new resource group if you don't already have one available. Using a parameter file, you can deploy your ARM template with the following commands:

PowerShell:

Click here to view code image

```
C:\> New-AzureResourceGroupDeployment -Name ExampleDeployment
-ResourceGroupName ExampleResourceGroup -TemplateFile
<PathOrLinkToTemplate> -TemplateParameterFile
```

```
<PathOrLinkToParameterFile>
```

Azure CLI:

Click here to view code image

```
azure group deployment create -f <PathToTemplate> -e
<PathToParameterFile> -g ExampleResourceGroup -n ExampleDeployment
```

Deploying an ARM Template from Version Control

We can deploy directly from a publicly accessible Git repository by providing the template's URI:

Click here to view code image

```
azure group deployment create --template-uri <repository item URI>
```

For example, the Azure CLI command for deploying a new storage account using the Azure GitHub repository would look similar to the following:

Click here to view code image

```
azure group deployment create --template-uri
https://raw.githubusercontent.com/Azure/azure-quickstart-templates/
master/101-create-storage-account-standard/azuredeploy.json
```

For more information on Azure Resource Manager, including guidance and best practices, take a look at the product documentation at https://azure.microsoft.com/en-us/documentation/articles/resourcegroup-overview/.

Multivendor Provisioning

The Azure Resource Manager (ARM) is the recommended approach to provisioning infrastructure resources in Azure. Sometimes there is a need to provision all or some of the resources for an application outside Azure. These can be resources in a private cloud, another public cloud, or even some other hosted services like the hosted Elasticsearch services offered by Found (http://found.no).

There are a lot of different ways to approach this. We could use a combination or custom scripts for provisioning the other resource and then deploying the Azure resources through ARM using either the declarative or imperative model. We could transform the ARM templates as needed, or even have an ARM template invoke custom provisioning scripts to configure these resources. In addition to those options, there are some other infrastructure provisioning orchestration tools available in the market that enable us to define our infrastructure in a cloud platform vendor-agnostic way, and then use cloud-specific providers to do that actual work of provisioning. An example of one such tool is Terraform (http://terraform.io). Terraform is capable of provisioning Azure infrastructure through an ARM-based provider, along with some other popular cloud platforms.

Terraform, by HashiCorp, is a tool for building, changing, and versioning infrastructure in a safe, efficient, and consistent way. Using configuration files that describe the final state of an application, Terraform generates and runs an execution plan that builds the necessary infrastructure. Terraform can be used to manage tasks as small as individual computing resource allocations or as large as multivendor cloud deployments.

Terraform integrates with different resources through the use of providers. Providers form a bridge between the Terraform configuration and any underlying APIs, services, or platforms. The Azure provider enables you to provision and manage Azure resources, including virtual machine instances,

security groups, storage devices, databases, virtual networks, and more. This tool can be useful when you need to define and provision infrastructure on multiple cloud platforms.

Scheduling and Cluster Management

We have discussed some of the tools and methodologies used to provision our infrastructure; the virtual machines, networking, storage, and any other Azure services used by our applications. Now we need to schedule our services to run on the virtual machines that have been provisioned.

Scheduling is seemingly simple. For example, find and deploy five instances of my payment service in the cluster. However, this simple task can quickly become complicated as we introduce dependencies, constraints, optimizations, different types of resources, and requirements for those resources. A scheduler needs to take all these factors into consideration when determining the most optimal resources to deploy our services to.

Challenges

Scheduling services can quickly become complicated, especially as our application and infrastructure grow. We need to balance efficiency, isolation, scalability, and performance, all while accounting for each application's various requirements. We need a service that can automate the decisions and all the little complexities involved in determining what machines in our cluster should run each instance of a service.

Efficiency/Density

Efficiency measures how well our infrastructure schedules services based on the resources available. In an ideal environment, services would be evenly distributed across multiple servers and there would be no wasted resources.

In the real world, each service has its own unique resource requirements, and all nodes might not provide the same resources. A service can have low processor and low memory requirements, but it can require lots of storage. Another service can have a requirement for low amounts of high-throughput storage—in other words, it would require solid state storage or a RAM disk instead of a traditional hard drive.

The scheduler needs to quickly identify optimal placement of our services alongside other services on nodes in the cluster. On top of this, the scheduler needs to constantly account for changing resources due to hardware provisioning and node failures.

Isolation

Contrary to the distributed nature of scheduling, services make heavy use of isolation. Our services are designed to be created, deployed, and destroyed repeatedly without affecting the performance or availability of other services. Although services can communicate with each other, removing isolation and creating dependencies between them essentially defeats the purpose of a microservices architecture.

As an example, container-based solutions such as Docker use the Linux kernel's cgroups feature to control resource consumption by specific processes. They also make use of kernel namespaces, which limits the scope of a process. This can greatly improve fault and resource isolation of services in a microservices architecture. In the event of an unexpected failure, a single service would not

compromise the entire node.

Scalability

As application complexity grows, so does the complexity of the data center. Not only do we need to design our infrastructure around existing services, but we also need to consider how our infrastructure will scale to meet the demands of future services. The scheduler might need to manage a growing number of machines. Some are even able to increase and decrease the pool of virtual machines to match demand.

Performance

Performance problems can be indicative of a poor scheduling solution. The scheduler has to manage an extremely dynamic environment in which resources are changing; the services running on those resource are changing, and load on the services are changing all the time. This can be complex, and maintaining optimal performance can often require a great monitoring solution.

Identifying the most optimal resource for a task can take time and sometimes it's important that the task is scheduled quickly to respond to an increase in demand or a node failure.

A Scheduling Solution

As we just learned, the simple job of scheduling tasks to run on a node can quickly become complex. Resource schedulers in operating systems have evolved over time and we have had a lot of experience scheduling local resources. Solutions for scheduling meant to work across a cluster of machines is not nearly as mature or advanced. However, some great solutions are available to help in addressing this need, and they are evolving at a very rapid rate.

Many of the cluster scheduling solutions today provide a lot of similar features and approaches to container placement. Before covering some of the more popular schedulers, we will first discuss some of the common features available in some of these schedulers.

Strategies, Policies, and Algorithms

The strategies, or logic, used for determining the nodes on which to run a task or service can be as simple as selecting a random node that meets our constraints. Or it can be something far more complex. As mentioned earlier, we would like to optimize resource utilization without adversely impacting application performance. Some strategies a scheduler might use are bin packing, spread, and random. A bin packing approach simply fills up one node and then moves on and starts filling up the next. A spread approach essentially performs a round-robin placement of services across all available nodes. A random approach will randomly place services across nodes in the cluster. It can get a lot more complicated than this as we place different types of workloads next to one another and need to prioritize them.

Many of the technologies used for scheduling will offer extensibility for scheduling strategies, so that more complex strategies can be used if needed, or strategies that better meet the business requirements.

Rules & Constraints

Most schedulers enable us to tag nodes and then use that data to apply constraints when scheduling tasks. For example, we could tag nodes that have SSD storage attached to them, and then when we are

scheduling tasks, we can provide a constraint stating that the task must be scheduled to a node with SSD storage. We can tag nodes with fault domains and provide a constraint stating that the scheduler needs to spread a set of tasks across the fault or update domains, ensuring all our service instances are not in the same fault domain.

> **Availability Sets**
>
> In Azure, virtual machines are organized into availability sets which assign fault and update domains to the virtual machine instances. Update domains indicate groups of hardware that can be rebooted together. Fault domains indicate groups of machines that can share a common power source and network switches, and so on.

Dependencies

A scheduler needs a way to define a grouping of containers, their dependencies, and connectivity requirements. A service can be composed of multiple containers that need to remain in close proximity. Kubernetes enables you to group a set of containers that make up a service into pods. A good example of this is a node.js service that uses NGINX for its static content and Redis as a "local" cache. Other services might need to be near each other for efficiency reasons.

Replication

A scheduler needs to deal with optimally scheduling multiple instances of a service across the cluster. It's likely we don't want to place all the instances of a service on a single node or in the same fault domain. The scheduler often needs to be aware of these factors and distribute the instances across multiple nodes in the cluster.

Reallocation

Cluster resources can change as nodes are added, removed, or even fail. In addition to the cluster changing, service load can change as well. A good scheduling solution needs to be able to constantly monitor the nodes and services, and then make decisions on how to best reallocate resources based on availability as well as the most optimal performance, scale, and efficiency needs of the system.

> **Azure Healing**
>
> In Azure, if the hardware running the services fails, Azure will handle moving the virtual machine the services are running on. This situation, where a machine goes away and comes back, can be challenging for schedulers to manage. Also note that Azure works at the machine level and will not help us if an individual container fails.

Highly Available

It could be important that our orchestration and scheduling services are highly available. Most of the orchestration tools enable us to deploy multiple management instances, so that in the event one is unavailable for planned or unplanned maintenance, we can connect to another. Depending on the application requirements and cluster management features, it can be okay to run a single instance of that management and orchestration service. If for some reason the management node is temporarily

unavailable, it can simply mean we are unable to reschedule services until it's available again, and the application is not impacted.

Rolling Updates

As we roll out updates to our services, the scheduler might need to scale down an existing version of a service while scaling up the new version and routing traffic across both. The scheduler should monitor the health and status of the deployment and automatically roll back the update if necessary.

Autoscaling

The cluster scheduler might be able to schedule instances of a service based on time of day, or based on monitoring metrics to match the load on the system. In addition to being able to monitor and schedule task instances within a cluster, there could be a need to automatically scale the cluster nodes to reduce the cost of idle resources or to respond to increased capacity needs. The scheduler can raise an event for the provisioning systems to add new nodes to the cluster or remove some nodes from the cluster.

API

Continuous Integration/Deployment systems might need to schedule new services or update an existing service. An API will be necessary to make this possible. The API can even expose events which can be used to integrate with other services in the infrastructure. This can be useful for monitoring, scheduler tuning, and the integration of other systems or even cloud platform infrastructure.

There are a growing number of technologies available in the market that are used for the orchestration and scheduling of services. Some of them are still not necessarily feature-complete, but they are moving extremely fast. We will cover a few of the more popular ones in use as of this writing. The first we will look at is Docker Swarm.

Docker Swarm

Docker, Inc. provides a native clustering solution called Docker Swarm. A Swarm cluster is a pool of Docker nodes that can be managed as if they were a single machine. Swarm uses the standard Docker API, which means existing tools are fully compatible, including the Docker client. Docker Swarm is a monolithic scheduler where a single Swarm Master that's aware of the entire cluster state is responsible for scheduling.

As shown in Figure 5.2, a Docker Swarm cluster will contain one or more Swarm Masters, a number of nodes running the Docker daemon, and a discovery backend, not to be confused with container discovery services covered later in the networking section of this chapter.

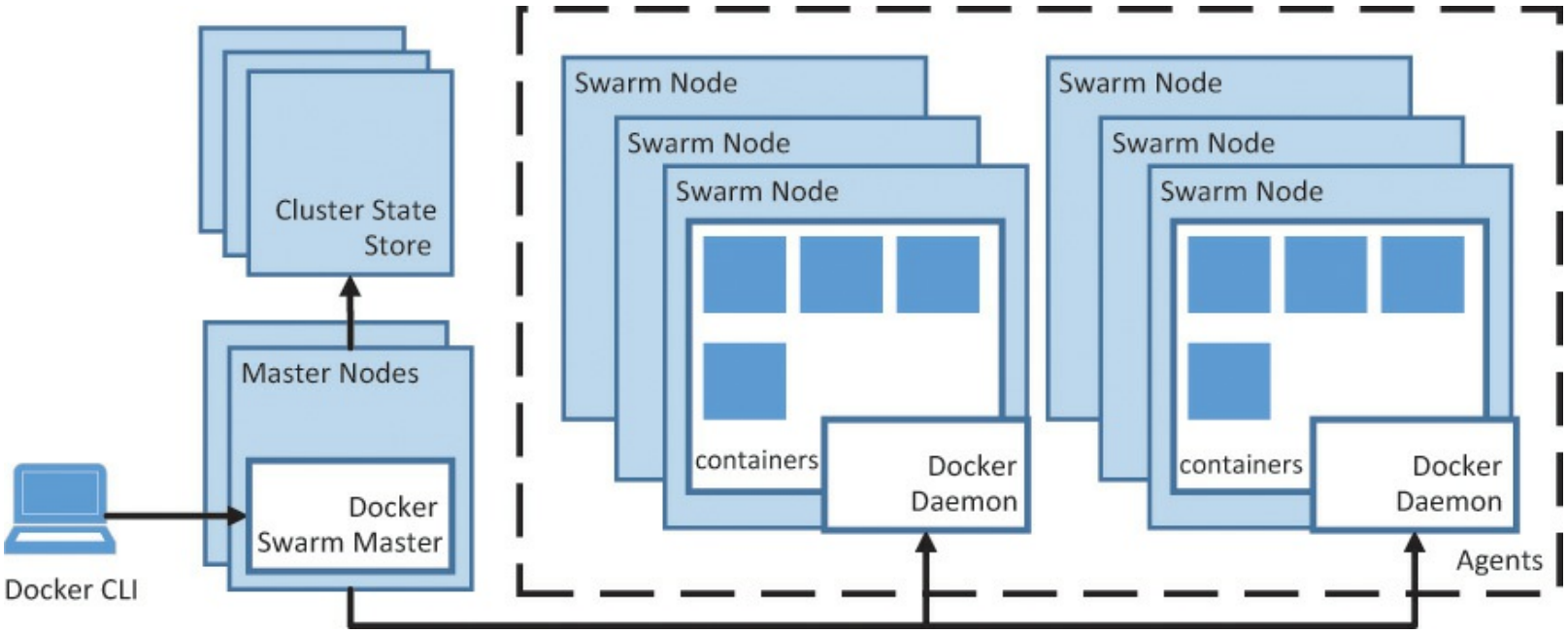

FIGURE 5.2: Docker Swarm cluster overview

Master Nodes

A Swarm cluster will contain one or more master nodes. Only one master node will be performing cluster scheduling work, but additional nodes can be running to provide high availability. The masters will elect a leader, and if for some reason the elected master is unavailable another master will be elected and take over the task of handling client requests and scheduling work. A supported service that can provide the necessary leader election feature is required. Services like Consul, Zookeeper, and etcd are commonly used for this purpose, as well as the discovery backend. The master uses the list of nodes in the discovery backend to manage containers on the nodes. The master communicates with the nodes using the standard Docker protocol, the same one the client uses.

Discovery Backend

The Docker Swarm discovery backend is a pluggable service that is used as a cluster discovery mechanism and will manage cluster state. When a node joins the cluster it will use this service to register itself and join the cluster. The discovery backend is pluggable, with a cloud-hosted backend available, as well as many other options.

Swarm Strategies

The Docker Swarm scheduler supports multiple strategies that determine how Swarm computes ranking for container placement. When we run a new container, Docker Swarm will schedule it on the node with the highest computed ranking for the selected strategy. Swarm currently supports three strategies; spread, binpack, and random. The spread and binpack strategies consider the nodes available CPU, RAM, and number of running containers. The random strategy simply selects a node and is primarily used for debugging. The spread strategy optimizes for nodes with the least number of containers, trying to keep a more even distribution. The binpack strategy optimizes for nodes with the most containers, attempting to fill nodes up.

An API is available for creating new Swarm strategies, so if a strategy with a needed algorithm does not exist, we can create one.

Swarm Filters

Docker Swarm comes with multiple filters that can be used to scheduler containers on a subset of nodes. Constraint filters are key/value pairs associated with nodes, and can be used for selecting specific nodes. Affinity filters can be used to schedule containers that need to be close to other containers. Port filters are used to handle host port conflicts, and are considered unique filters. A health filter will prevent scheduling of containers on an unhealthy node.

Creating a Swarm on Azure

A Docker Swarm cluster can be created on Microsoft Azure using an ARM template, the Azure Container Service (ACS), a Docker hosted service called tutum (http://tutum.co), or even Docker machine. Docker machine is a tool used to install Docker hosts on local machines or on cloud providers like Microsoft Azure.

An ARM template is currently available on GitHub to simplify the process of creating a Docker Swarm deployment on Azure (https://github.com/Azure/azure-quickstart-templates/tree/master/docker-swarm-cluster). This template can be copied and placed in source control management system for the project it will be used in and extended to meet the infrastructure requirements of the project.

The Docker Swarm Cluster Template will automatically build and configure three Swarm Manager nodes to ensure maximum availability. We can specify the number of application nodes via a template parameter. If we need to modify the template it can easily be forked and modified to meet the needs of the project. For example, we might want to install some additional monitoring services on the nodes, change how the network is set up, or use a different cluster discovery back end.

Connecting to the Swarm

Once the cluster deployment has completed we can connect to Docker Swarm in the same way we connect to a single instance of the Docker daemon. We can use the Docker client to execute standard Docker API commands such as `info, run, and ps.`

For example, look at the following command lists running containers on an Azure Swarm:

Click here to view code image

```
$ docker -H tcp://<manager DNS name>-manage.<location>.cloudapp
.azure.com:2375
```

> **Secure Communications**
>
> For situations that require the Docker client to connect to a daemon over a public network, it is recommended that Transport Layer Security (TLS) is configured. More information on configuring TLS in Docker can be found here: https://docs.docker.com/engine/security/https/. Alternatively, an SSH session can be established to the Swarm Manager and the local Docker client can be used.

One of the nice things about working with Docker Swarm is the simplicity. The Docker Swarm API is like working with a single instance of Docker, only there is an entire cluster behind it.

Kubernetes

Similar to Docker Swarm is Kubernetes (often abbreviated as k8s), an open-source Google project for managing containers in a cluster.

Kubernetes acts as a framework for deploying, scheduling, maintaining, updating, and scaling microservices. In essence, it abstracts the process of deploying the service from the user and actively monitors the state of the application infrastructure to ensure integrity. If a problem occurs, Kubernetes can automatically rebuild and redeploy a failed container.

A key difference between Kubernetes and Docker Swarm is how they treat clusters of containers. With Docker Swarm, individual containers are unified under a single API. With Kubernetes, containers are grouped into logical units which are then managed, monitored, and assigned resources by the cluster master. One of the key features of Kubernetes is that it enables users to define the ultimate state of the cluster, while managing the cluster's various components to match that state.

In addition to the Scheduler, the Kubernetes master includes an API Server and a Controller Manager as we see in Figure 5.3. The kubectl client talks to the API Server on the master node and sends a configuration for a replication controller, with a pod template and desired replicas. Kubernetes uses this information to create a number of pods. The scheduler then looks at the cluster state and schedules work on the nodes. The kubelet, an agent running on each node, monitors changes in the set of pods assigned to it and then starts or kills pods as needed.

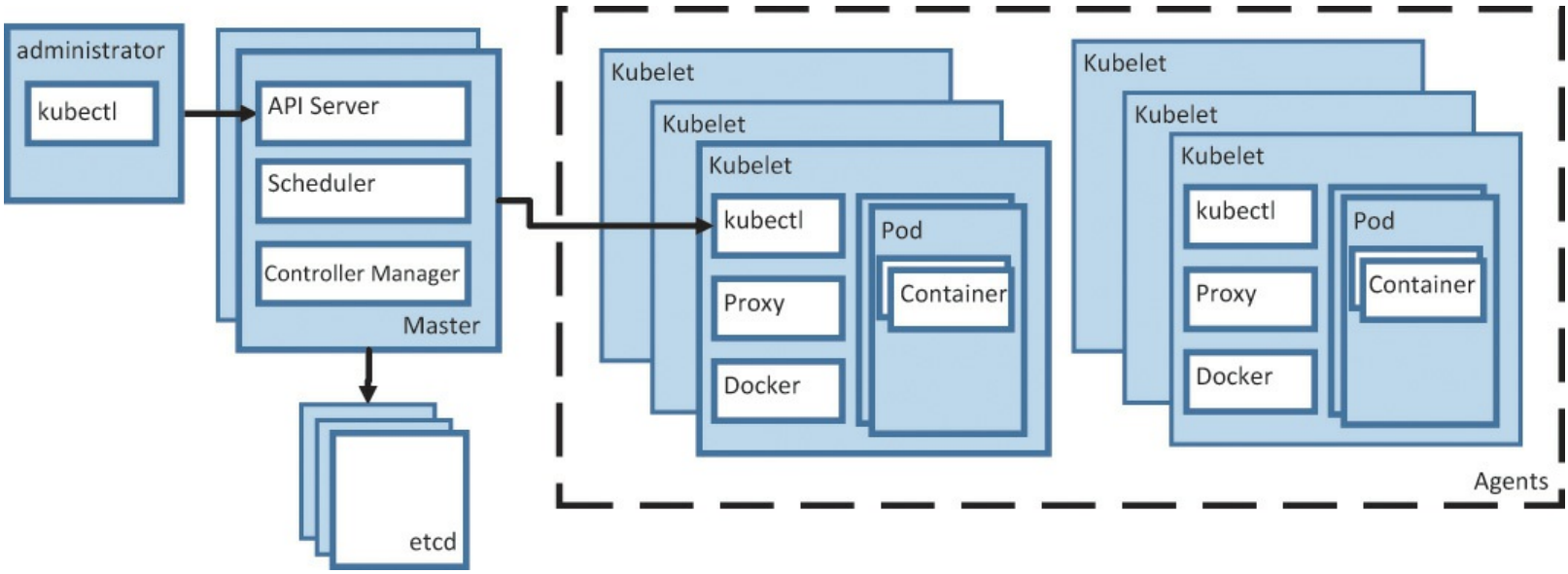

FIGURE 5.3: Kubernetes cluster overview

Components

A basic Kubernetes cluster consists of node agents (known as kubelets) managed by a Kubernetes Control Plane running on a master node.

Pods

Kubernetes groups related containers into logical units called *pods*. Pods are the smallest components that can be created, deployed, or managed by Kubernetes. However, they generally consist of containers that perform complementary services. For example, a pod that provides a website could consist of a web server container and data storage container. Containers within a pod can view other containers' processes, access shared volumes, and communicate via message queues.

Replication Controllers

When running multiple instances of a pod, Kubernetes controls the number of instances by using a *replication controller*. A replication controller ensures that a specified number of pods are available at any given time. For instance, if you have two instances of a pod running for load balancing purposes and an instance fails, a replication controller will automatically start a new instance.

Labels and Selectors

A *label* is metadata attached to Kubernetes objects such as pods. A label consists of a key/value pair that can be attached, modified, and removed at any point during the lifetime of the object. Labels are generally used to organize and group various objects. A label *selector* enables us to group sets of objects based on the value of a label.

Services

Similar to how pods define a set of related containers, *services* define a set of related pods. Services are a stable abstraction in the cluster that provide routing, service discovery, load balancing, and zero downtime deployments. Applications consuming the service can use either the host name or IP address of the service and the requests will be routed and load-balanced across the correct pods.

When a service is created, Kubernetes assigns it a unique IP address, and although pods will come and go, services are more static.

Volumes

Containers are stateless by design. When a container is deleted, crashes, or refreshed, any changes made to it are lost. With *volumes*, containers can preserve data in a directory that exists outside of the container. Volumes also enable containers to share data with other containers in the same pod. Kubernetes supports several types of volumes, including local folders, network-attached folders, cloud storage volumes, and even Git repositories.

Names and Namespaces

Each object in Kubernetes is identified by an identifier and a *name*. Unlike labels, names and IDs are unique for each object. Names can be provided, but an ID is generated by Kubernetes.

Namespaces enable you to create multiple virtual clusters on the same physical cluster. Objects that exist in the same namespace are restricted to that namespace; they can communicate with each other, but not with objects in other namespaces. There are some exceptions, such as nodes, persistent volumes, and namespace objects themselves.

Other Components

Kubernetes supports other features such as annotations for attaching non-identifying metadata to objects, secrets for storing sensitive data, and more. For additional information, see the Kubernetes User's Guide at http://kubernetes.io/v1.1/docs/user-guide/README.html.

Kubernetes on Azure

Kubernetes can be deployed on Microsoft Azure, and a set of scripts for easily deploying to Azure are maintained in the Kubernetes project. These scripts can be found on the Kubernetes site (https://kubernetes.io) in the Getting Started section of the documentation.

Apache Mesos

Apache Mesos is an open-source cluster manager and scheduling framework for pooling and sharing resources like CPU, memory, and storage across multiple nodes. Instead of having a specific node for a web server or database server, Mesos provides a pool of machines for running multiple services simultaneously.

In Figure 5.4 we see an Apache Mesos Master with a standby node for high availability. Zookeeper is used for cluster state and leader election, and frameworks are used for scheduling work in the cluster.

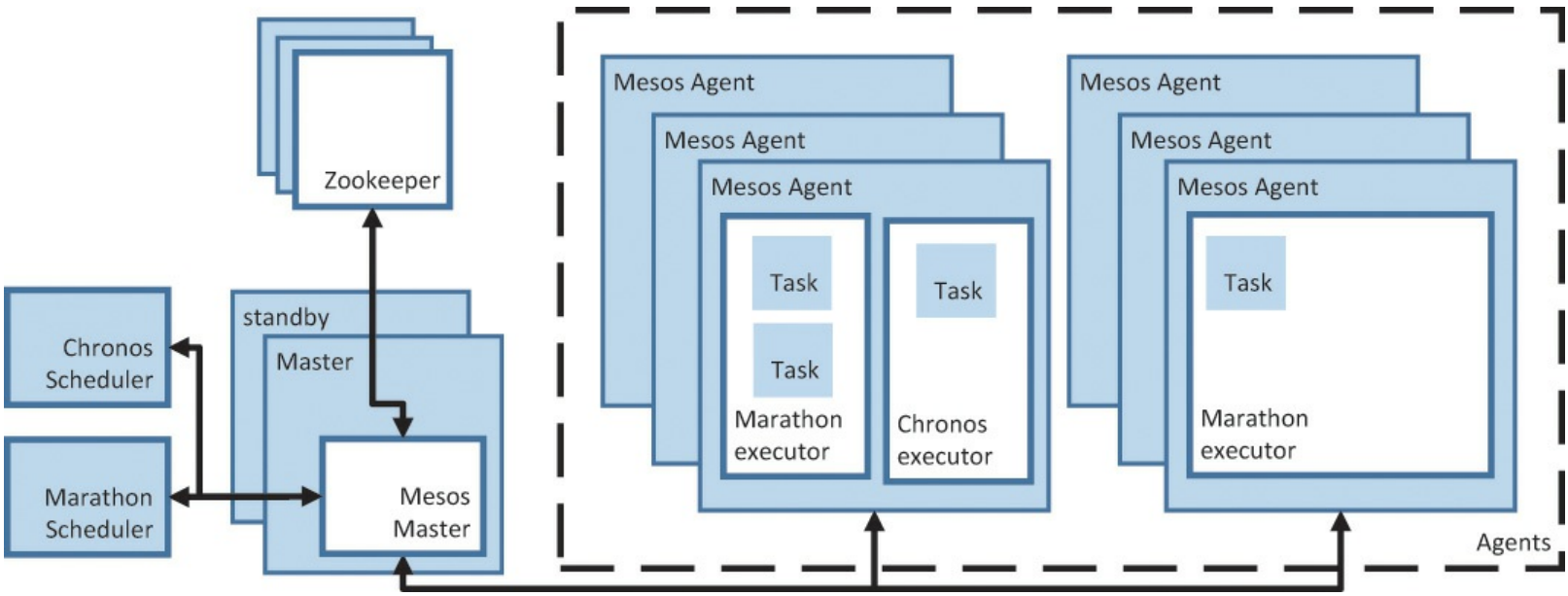

FIGURE 5.4: Mesos cluster overview

Core Components

An Apache Mesos cluster consists of three major components: masters, agents, and frameworks.

Masters

A *master* daemon manages one or more agent daemons. It's responsible for tracking each agent's available resources, tracking active applications, and delegating tasks to agents.

Agents

Agent daemons run on each cluster node. Agents cluster nodes to provide the physical resources that the master tracks, pools, and distributes to each application/framework.

Frameworks

Frameworks are applications that run on an Apache Mesos cluster. Each framework is split into tasks, which are individual work units that run on Apache Mesos agents. Unlike many clustering solutions, Mesos enables frameworks to control much of the delegation of tasks. When Apache Mesos allocates resources to a framework, it makes an offer to the framework; if the offer doesn't satisfy the framework's constraints, the framework can reject it. Frameworks can be configured to accept or reject resources based on certain constraints.

Frameworks themselves consist of two components: schedulers and executors. Schedulers register with the master and handle resource offers. Executors are processes on the agent nodes that run tasks.

Mesosphere DCOS

Mesosphere Data Center Operating System (DCOS) is a commercial product from Mesosphere, Inc. (http://mesosphere). DCOS is based on Apache Mesos and includes enterprise-grade security in addition to many other core system services, like Mesos-DNS, a Command Line Interface (CLI), an API, and more.

Marathon

Marathon is an open-source framework from Mesosphere's DCOS, and is used for long-running tasks. The microservices we will deploy are generally long running, and we will need to ensure they continue running until we decide we need to stop them for some reason, like reducing the count or updating them with a new version. Marathon can be used to ensure a defined number of our services remain running in the cluster, even if a node on which one of our service instances is running goes away—provided there are enough resources, of course. In addition to scheduling containers, marathon can schedule and run commands in the cluster. For example, we could bootstrap a node.js application that pulls down the necessary artifacts and runs. Marathon comes with a UI and an API that can be used to schedule tasks.

We will actually use Marathon to deploy and run our sample application in a Microsoft Azure Container Service.

Chronos

Chronos is an open-source framework from Mesosphere's DCOS, and is used as a replacement for running cron jobs in a cluster. It's a distributed, fault-tolerant scheduler that supports custom Mesos executors as well as default command executors. Chronos does a lot more than cron, and can schedule jobs that run inside Docker containers, use repeating interval notation, and can even trigger jobs by the completion of other jobs. Chronos includes a UI and API that can be used to schedule tasks.

Almost all large distributed applications include a growing number of scheduled jobs based on time or interval.

Using Apache Mesos to Run Diverse Workloads

Apache Mesos provides a platform for managing a cluster, using frameworks, and running tasks. Whereas we can run frameworks for Hadoop, Spark, Storm, and Jenkins, we can also use Marathon to run specific commands. For example, we can use Apache Mesos to launch Docker containers, as well as workloads like Spark, Cassandra, or Kafka alongside Docker Swarm on the same cluster.

The Kubernetes and Swarm teams have also created Mesos frameworks for launching Kubernetes or Swarm on Mesos. As a framework, this offers much better integration than using Marathon to launch a Docker Swarm cluster on Mesos. Even though you can use Marathon or Aurora to schedule Docker containers, there can be some benefits to deploying Docker Swarm on Mesos. Docker Swarm exposes the same API used by developers to run containers, making it easy to get started, and they continue to add new features like software-defined networking that can be leveraged.

Using Apache Mesos to manage our application services removes the need to provision servers for specific services. Nodes are merged under a unified, logical interface. This makes it easy for frameworks like Marathon applications to assign themselves to the necessary resources without

having to worry about load balancing, redundancy, scalability, or integrity.

Service Discovery

In more traditional distributed systems, services run at fixed, well-known locations, and there are generally fewer dependent or collaborating services. A microservices-based application often runs a large number of services in an environment where the service locations and instances are very dynamic. As we discussed in the resource scheduling section, services are often deployed in a shared cluster of virtual machines. Services are scaled up or out across a cluster of resources to meet the demand. Nodes come and go, and services are reallocated to optimize resource usage. Services provisioned in these environments require some mechanism to enable them to be easily discovered for requests to be sent to them.

Our services will need to communicate with each other, and inbound requests into the system need to be routed to the appropriate services. For this we simply maintain a list of services and their locations, so that we can look up the endpoint location when we need to call it. This is referred to as service discovery, which can be implemented a number of different ways, using a growing number of technologies.

Figure 5.5 shows a conceptual service discovery service with service announcement and lookups.

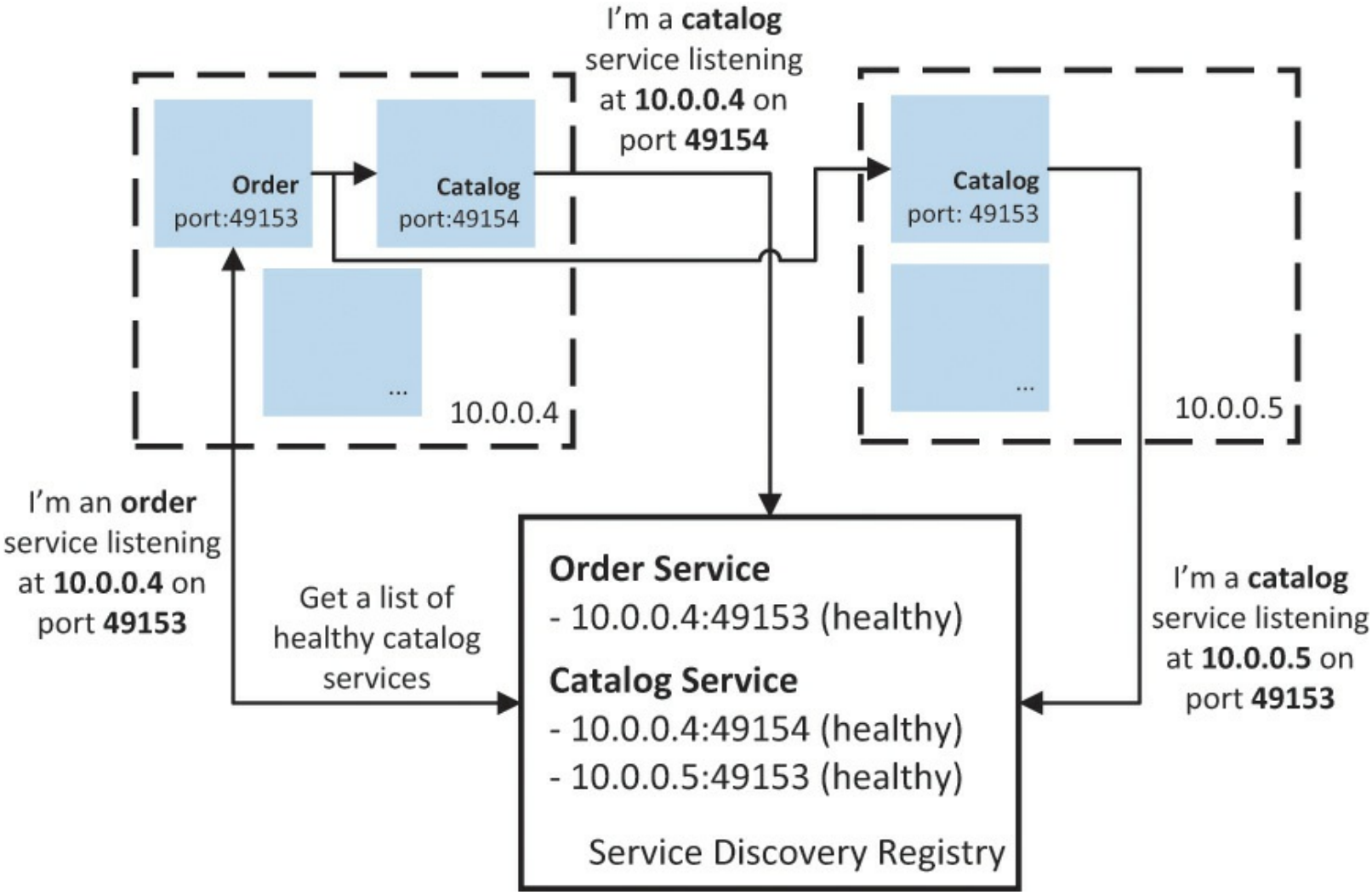

FIGURE 5.5: Service discovery announcement and lookup

Service Discovery Components and Flow:

- **Announcements:** Every service instance registers with the service discovery registry and can

be responsible for sending updates at regular intervals: “Hello, I’m a ‘Catalog Service’ and I’m ready for requests at IP address N.N.N.N on port NNN,” and might subsequently send a follow-up: “Hi, I’m still here and healthy.”

- **Registry:** Manages the list of service instances by service and can perform routine service checks or remove a service instance form the list if it does not check back in, assuming it is no longer with us. “Are you still healthy, ‘Catalog Service’?” “200 I’m still healthy.”
- **Lookups:** When the “Order Service” needs to call a “Catalog Service,” it will get a list of healthy “Catalog Service” instances to send the request to. “Give me a list of ‘Catalog Services.’” Maybe the list is cached in the node for some time and then load balanced across the list of “Catalog Service” instances to distribute the load.

This is generally how service discovery works, even though there are a number of different ways to implment this, and always some trade-offs that need to be considered. Let’s have a closer look at sevice registration, lookup, and service discovery stores.

Service Registration

Also known as service announcements, this is simply the process where changes to a service status are written to a service registry. These changes are typically when a service is available to handle requests, or is no longer available to handle requests. When a new service instance has been scheduled on a node and is ready to receive requests, we need to enable consumers of the service know that it’s ready and where it is. Multiple instances of a service can be scheduled across the cluster, all listening on different IP addresses and ports. It’s part of a service announcement’s job to simply report to the service registry, “I’m an Order service ready to receive requests and I am listening on 10.0.0.4 at port 9001.” The service registry then maintains this information, and when the service is shut down the service needs to let the registry know that it’s going away. We can also configure a time-to-live (TTL) on the records and expect the service to routinely check back in, or we can perform health checks on the service to make sure it has not silently disappeared.

A lot of different ways exist to implement this seemingly simple task. Common approaches are to make this the job of the service orchestration tools, make it part of the service initialization, or include it as a node management sidecar process. Kubernetes, for example, will manage this information in a service registry store like etcd when scheduling pods to run on nodes. Projects like Airbnb Nerve or Glider Labs Registrator will automatically register and deregister services for Docker containers by inspecting containers on a node as they come online. Another approach would be to have the service register itself as part of the service startup, and to implement this in a framework.

Service Announcement Implementation Options:

- Orchestration tooling is responsible for maintaining this information. The orchestrator will maintain a list of scheduled and running service instances in the cluster as well as the health of each instance.
- The service itself implements announcements, and on service startup it will make a call to register itself. This can be a simple approach and it can be implemented in a shared library for each supported language.
- A sidecar service running on each node monitors services and performs the registration and deregistration. Externalizing registration to a sidecar process means the service implementation

does not need to be concerned with the task of registration or deregistration.

Service Lookup

Also referred to as service discovery, is simply the process of finding services and their related endpoint information. Clients and load balancers need to be able to look up the location of services to send requests. When one service needs to call another service, it will look up endpoint information, and quite possibly health status, when calling the service. Centralized load balancers will often monitor and read this information to maintain a list of back ends for routing and load balancing traffic across multiple instances of a service.

Service Lookup Implementation Options:

- *DNS-based lookups* can be an easy option that requires few changes.
- A *client framework* in the service can be used to look up endpoints and to load balance.
- A *local proxy* on each host can maintain lookup information and proxy requests to the correct service, as well as load balance requests.
- The client uses a *centralized proxy* load balancer that performs lookups. The client will then access the service through a shared load balancer that is responsible for maintaining a list of back-end service instances.

In Figure 5.6 we see a configuration that uses a local proxy, such as NGINX, to route requests. The order service talks to a well-known endpoint like localhost:9001 to call a dependent service, and the local proxy handles the lookup. This can simplify application design, moving the service lookup to a proxy. This is sometimes referred to as an arbiter.

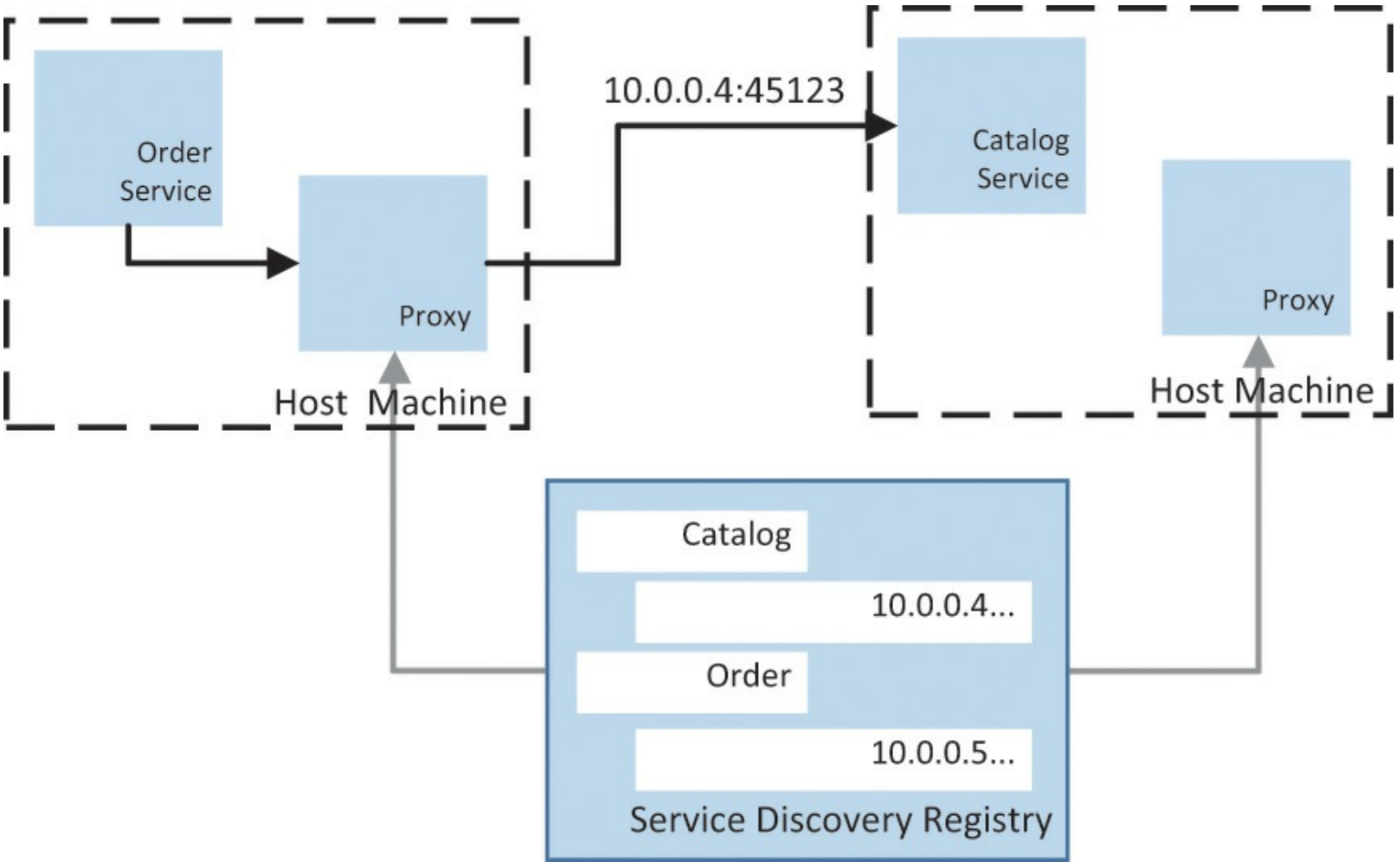

FIGURE 5.6: Diagram showing a local proxy in each host to perform service routing

This approach would also remove the need to develop language-specific frameworks for each language you need to support.

> **More than Just a Routing Proxy**
>
> The proxy used to route requests to the appropriate service instances could also provide resiliency implementations to handle concerns like circuit breaker and retry. This would eliminate the need to implement and maintain a library for every language used.

Service Registry

A service registry is simply a database that maintains a list of services, their endpoints, and quite often additional service metadata which is used when consuming the service. Some service discovery technologies, like Consul, will perform health checks and maintain health state of service instances.

Features

There are a number of different technologies commonly used as a service registry. Each technology offers a similar set of features. Let's first have a look at the features of a conceptual service discovery store, and then the common technologies used.

Highly Available and Scalable Store

One of the more important features is that the service registry is highly available and scalable. All our services will be using the service registry to either announce and/or perform service lookups. If we are unable to perform a lookup, then we might not be able to find the service we need to send a request to. Most of the service registries use multiple redundant nodes so that if one node goes down the application is able to connect to another.

Service Announcement and Lookup API

It's important that our service announcement implementations are able to register changes and that services are able to perform lookups. Service announcement and lookup client tools contain integrations with different technologies, and this can influence your decision as to which technology you select.

Notifications

When a service changes, such as when instances are added, removed, moved, and so on, it's sometimes necessary to be notified of these changes. Technologies like Confd (https://github.com/kelseyhightower/confd) or Consul Template (https://github.com/hashicorp/consul-template) can monitor changes and update configuration based on the data. This is useful when maintaining a list of back ends in something like NGINX or HAProxy.

Health Checks

Some solutions, like Consul, can be configured to perform health checks and maintain health metrics about the services that are registered. This can be used for monitoring and also for routing logic. Shared circuit breaker state might be maintained here, and if the service is having problems, the client

might not call it.

DNS Protocol Integrations

DNS has been around for a long time. It is used for name resolution. It's widely supported and easy to use. This can be used to provide lookup discovery with no implementation changes. Service registration can be exposed exclusively through DNS or as an additional alternative.

Technologies

Many of the technologies discussed here do more than just store service discovery data, and are often used for coordination as well as a configuration store. In addition to the features and libraries that integrate with these services, you should consider the following: If you are going to deploy Docker Swarm and use Consul for cluster coordination, then maybe you want to consider using it for service discovery so you don't have to deploy and maintain another system. You need to be careful and consider how this could affect the scalability and performance of the cluster, as well as the different requirements and features of the technology, like the consistency models.

> **Note**
>
> When sharing deployments of a service, be careful that one workload does not adversely affect the other. Using the Zookeeper deployment that Apache Mesos uses for maintaining cluster state as a service discovery store can impact the performance of Apache Mesos. In a production environment it is generally recommended that Apache Mesos should use a dedicated Zookeeper deployment.

DNS

The simplest solution would be to use DNS and create a name for each service; then clients can discover service instances using standard DNS. DNS for service discovery does have its limitations. First of all, DNS does not support pushing changes, so we need to poll for changes. Caching and propagation delays can cause challenges and latency in updating state. Some applications will even cache DNS when they start up.

If planning to use DNS for service discovery, consider some of the more modern DNS services. Mesosphere has created a project called Mesos-DNS which provides DNS-based discovery in Apache Mesos (https://github.com/mesosphere/mesos-dns). SkyDNS (https://github.com/kynetservices/skydns) is another popular option built on top of etcd. Consul is also capable of exposing service lookup through DNS as well as its standard API. The Docker and Weave networking features that are discussed further in the following overlay section also provide DNS-based container discovery.

DNS-based service discovery can be adequate and readily available in the platform, as with the Azure Container Service. When using DNS for the discovery of endpoints that are very dynamic, we need to take care in setting appropriate TTL and to understand how our clients are caching DNS. The argument against DNS is that it was designed for a different purpose and the caching can cause a lot of challenges in production.

Consul

Consul is an open-source project from the folks at Hashicorp, the same people that brought us Terraform, Vagrant, and Vault. Consul provides an HTTP API that we can use for service announcement and discovery as well as a DNS interface. In addition to service discovery, Consul can perform service health checks, which can be used for monitoring or service discovery routing. Consul also has a very nice templating client, which can be used to monitor service announcement changes and generate proxy client configurations. Other features include a key/value store and multiple data center support.

etcd

etcd is a distributed key value store from the folks at CoreOS. It provides reliable storage of data across a cluster of machines. It is often used as a service discovery store when building service discovery systems. It originated in the CoreOS project, is included in the distribution, and can be installed on other distributions.

Zookeeper

Zookeeper is an open-source Apache project, and provides a distributed, eventually consistent hierarchical key/value store. It's a popular option for a service discovery store when building service discovery systems. Mesos uses Zookeeper for coordination, and you might want to consider whether or not the technology is already used in your infrastructure when selecting your service discovery tool.

Eureka

Eureka is an open-source service discovery solution developed by Netflix. It's built for availability and resiliency, and is used at Netflix for locating services for the purpose of load balancing and failover of middle-tier servers. The project includes a Java-based client that offers simple round-robin load balancing. Additional application-specific metadata about a service can also be stored in Eureka.

Other Technologies

There are a number of other options that could be used for this purpose, but at the time of writing, these are the most popular. A number of factors will need to be considered when selecting a technology to use. You might already be using one of these to maintain cluster state or for leader election, or you can have a preference for another service that has better integrations with one of these.

Application/API Gateway

An application gateway, also commonly referred to as an API Gateway, is used in a microservices architecture for traffic aggregation and request routing of inbound client requests to the necessary services. These gateways are also quite commonly used for authentication offload and SSL offload, as well as quality of service throttling, monitoring, and many other things. Even very basic load balancing and routing requirements will often require a reverse gateway, as most load balancers in public cloud environments are not dynamic or flexible enough to meet the needs of modern microservices deployments.

Common Gateway Responsibilities:

- **Request routing and load balancing:** Perform service lookup and route requests to the appropriate service instances.
- **Request aggregation:** To reduce chattiness between the client and back-end services, the gateway will sometimes be responsible for handling a single request from the client, and then sending requests to multiple services and returning the aggregate results as a response to the client.
- **SSL offload:** A secure SSL connection is established between the client and the gateway, and then non-SSL connections are used internally, or different internal certificates are used to encrypt communications.
- **Authentication:** A gateway can be responsible for authenticating requests, and then passing customer information to the services behind them.
- **Protocol transformation:** The gateway can be responsible for converting protocols the client might need to those used by the internal services.

NGINX and HAProxy are two of the more popular technologies used as application gateways today. Both of these applications support dynamically configurable back-end pools. Tools like confd or Consul Templates can be used to maintain the back-end pools, replicating data from the service registry into configurations.

Figure 5.7 shows two or more instances of an application gateway running behind an Azure load balancer for high availability. Services will announce changes and the endpoint information to the service discovery registry. The gateway will update its upstream pools based on the services and service instance information. The gateway can then route requests using the path, so that requests to the "/catalog" path are routed to instances of the catalog service. The gateway can load balance across multiple instances of the catalog services, even though they are different machines and ports. Requests can be routed based on other factors such as the destination port, IP address, or some header value.

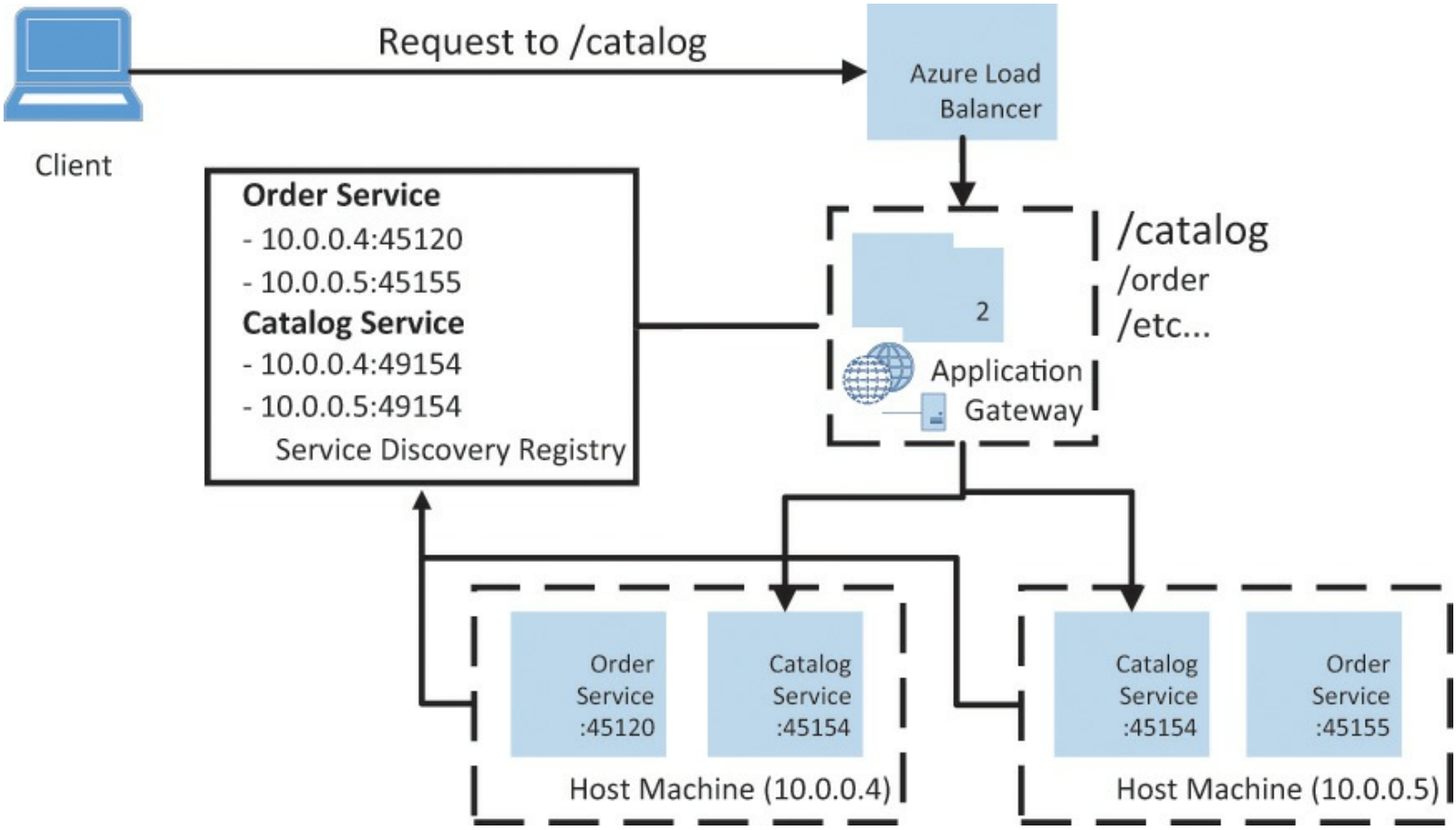

FIGURE 5.7: Application/API Gateway

We don't necessarily need to deploy our gateway onto separate dedicated nodes. We could instead configure the Azure load balancer to route traffic to a couple of nodes in the cluster, and then use tagging and constraints to deploy a gateway container to appropriate hosts. We could also configure the load balancer to route traffic to every node in the cluster and set up a gateway on every node in the cluster. A gateway on every node in the cluster can work well for small clusters, but as the cluster size increases you might want to use a dedicated set of nodes. These nodes can be part of the cluster or they could be special virtual machines outside the cluster manager, even in a different edge network.

Deployment Considerations:

- **Dedicated gateway:** Use a dedicated set of gateway virtual machines managed independently of the scheduler. The Azure load balancer is then configured to load balance across this specific set of machines.
- **Peer gateway request routing:** With a smaller cluster, we can place a gateway on each node in the cluster and route requests. By doing this we can save management and compute costs by eliminating a couple of virtual machines. When adding a node to the cluster, we will need to update the load balancer.
- **Dedicated cluster nodes:** We can configure the Azure load balancer to route traffic to a subset of the nodes in the cluster and tag the nodes appropriately so that we can schedule a gateway to run on them. We will need to ensure the Azure load balancer configuration and node configuration are in sync.

Overlay Networking

It would be nice if all the containers across the cluster were addressable with their own IP address and we didn't have to deal with dynamic ports or port conflicts. Kubernetes even requires each pod to have its own IP address, which is assigned from a range of IPs on the node the pod is placed. We still need some service discovery mechanism to know what IP address the service instance is at, but we don't have to worry about port conflicts on the hosts. In some environments, it's not possible to assign enough IP addresses to a host, and we need to manage the assignment of the IP address ranges in the environment.

We can create an overlay network on top of the existing infrastructure network that can route requests between containers that are distributed across multiple nodes. This enables us to assign an IP address to each container and connect to the service using multiple and standard ports. This reduces the complexity of port mapping and the need to treat ports as a resource on host machines when scheduling work in the cluster.

Benefits:

- **Basic DNS use:** We can use DNS features to find containers in the network, and thus do not need to write additional code to discover the assigned host ports the service is running on.
- **Avoids host port resource conflicts:** This can eliminate port conflicts in situations where we might want to schedule tasks on a node needing to expose the same port.
- **Simpler support for connectivity with legacy code:** In some situations legacy code can make it difficult or nearly impossible to use a different port.
- **Networking management:** Although it is necessary to configure and manage an overlay network, it can often be easier managing the deployment and configuration of an overlay network than dealing with IP host ranges and service port mappings.

An overlay network can be extremely useful, especially with a large cluster or clusters spanning multiple data centers. There are, however, concerns with the additional performance overhead and the need to install and manage another service on the cluster nodes.

Figure 5.8 provides a visual representation of an overlay network created on top of the existing infrastructure network, enabling us to route traffic to services within a node. As we can see, the host machine is running a multi-homed gateway service that is bound to port 80 of the host machine on 10.0.0.4 and connected to the overlay network. The gateway service can proxy inbound requests to the order service on port 80 at 172.16.20.1, and the order service can connect to an instance of the catalog service at either 172.16.20.2 or 172.16.30.1 on well-known port 80. Any of the name resolution options can be used for service discovery including those that come with some of the overlay technologies we will cover here.

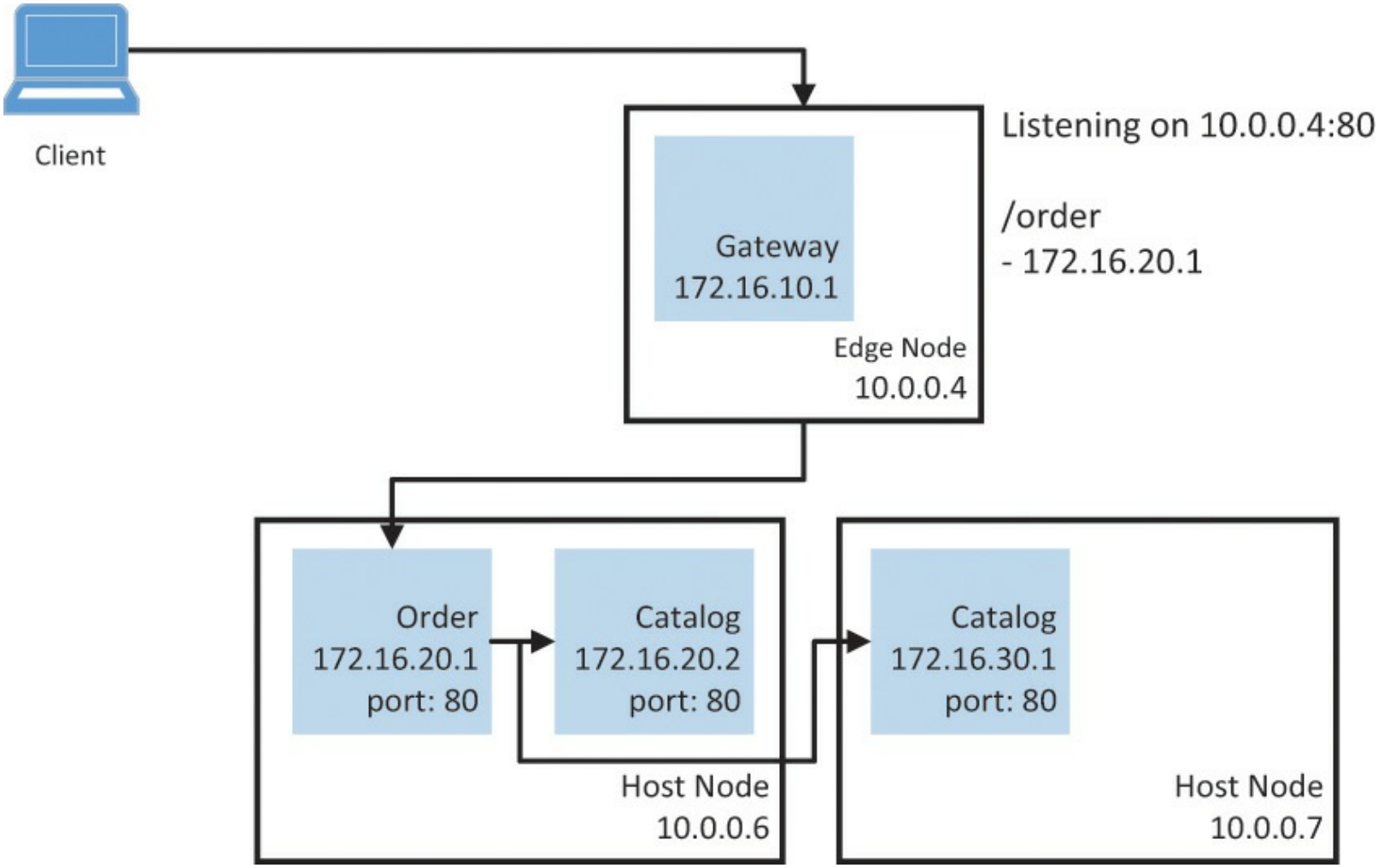

FIGURE 5.8: Service lookup using a proxy

The nice thing about this approach is that each container is now directly addressable, and although it can add a little complexity in the networking configuration, it significantly simplifies container lookup and management. All our services can be deployed and listening on well-known HTTP port 80 at their very own IP addresses. We can use well-known ports for the various services; it's easy to expose multiple ports on a container; and it can simplify service discovery.

Technologies

There are a number of technologies available in the market that can be used to create and manage an overlay network. These technologies all have trade-offs that need to be considered, providing a different set of features.

Docker Networking

The Docker engine includes a built-in multi-host networking feature that provides Software Defined Networking (SDN) for containers. The Docker networking feature creates an overlay network using kernel-mode Open Virtual Switching (OVS) and Virtual Extensible LAN (VXLAN) encapsulation. The Docker networking feature requires a key/value (KV) store to create and manage the VXLAN mesh between the various nodes. The KV store is pluggable and currently supports the popular Zookeeper, etcd, and Consul stores.

The Docker networking feature also provides service discovery features that make all containers on the same overlay network aware of each other. Because multi-host networking is built into the Docker engine, we would not have to deal with deploying a network overlay to all the host nodes. In

true Docker fashion, we can replace this with something like Weave, which offers more advanced features.

Weave

Weaveworks Weave Net (http://weave.works) is a platform for connecting Docker containers, regardless of where they're located. Weave uses a peering system to discover and network containers running on separate hosts, without the need to manually configure networking. Weave creates two containers on the host machine: a router container, which captures traffic intended for containers managed by Weave; and a DNS discovery container, which provides automatic DNS discovery for Weave containers.

Registering a Docker container with Weave assigns it a DNS entry and makes it available to other containers on the host. You can reference a Weave container from another simply by using its Weave-assigned DNS name. Weave also enables multiple containers to share the same name for load balancing, fault tolerance, hot-swappable containers, and redundancy. Weave also supports additional features such as encryption of traffic, host network integration, and application isolation.

There are some benefits to running Weave Net in place of the Docker networking. Weave Net does not require us to install and manage additional software, so there is no requirement for a KV store with Weave Net. It's more resilient to network partitions, offers simpler support for cross-site deployments, and provides a more robust service discovery option. It's also a great option for Kubernets, Mesos, and other container-centric schedulers.

Flannel

Flannel is a virtual mesh network that assigns a subnet to each container host. Flannel removes the need to map ports to containers by giving each host a pool of IP addresses, which can be allocated to individual containers. Flannel is a CoreOS project, but can be built for multiple Linux distributions.

A basic Flannel network encapsulates IP frames in a UDP packet. Flannel can use different back ends, including VXLAN, Amazon VPC routes, and Google Compute Engine routes.

Project Calico as an Alternative to Overlay

Project Calico (http://www.projectcalico.org) seeks to simplify data center networking by providing a purely IP-based solution. Working over both IPv4 and IPv6, Calico supports container-based solutions such as Docker, as well as OpenStack virtual machines.

With Calico, each container is given a unique IP address. Each host runs an agent (known as "Felix") that manages the details of each IP address. Using the Border Gateway Protocol, each host routes data directly to and from each container without the need for overlays, tunneling, or Network Address Translation. Access Control Lists can increase or limit public access to individual containers, isolate workloads, or implement security.

This approach can offer better scale and performance with reduced overhead in the VXLAN tunnels, and simplify network troubleshooting because the packets are not encapsulated.

Summary

In this chapter we covered a lot of material for building an environment on which to deploy and run our microservices applications. We covered cluster provisioning and options for scheduling services

in the cluster. We use discovery services for routing, and connectivity and possibly additional networking tools to meet our dynamic and configurable networking requirements. In the subsequent chapters we will cover monitoring the services to ensure they continue to operate as expected, as well as addressing configuration and management of the services.

6. DevOps and Continuous Delivery

One of the keys to the successful use of a microservice architecture is ensuring you have an automated, well-defined workflow where development and operations work together to produce agile, high-quality releases. This is the essence of DevOps. In this chapter, we'll provide an overview of DevOps, its benefits, and one of the most important facets of DevOps: building a culture of DevOps in your organization. Next we'll discuss creating environments in Azure for a continuous delivery (CD) pipeline and how a microservice is validated through a series of tests from code check-in to deployment in production. Finally, we'll discuss key criteria for choosing a continuous delivery tool.

DevOps Overview

DevOps is the combination of development and operations teams working together toward a unified goal: Ship the highest-quality code and infrastructure in the shortest span of time to deliver value to customers faster. With DevOps, operations are a core part of every step in the development pipeline. This includes the developers writing the code as well as the engineering teams that provision the hosting infrastructure and build and manage the release pipeline. It includes release engineers, database operations, network operations, security operations, and many others. The Microsoft model for DevOps shows how teams go through four phases, as shown in Figure 6.1.

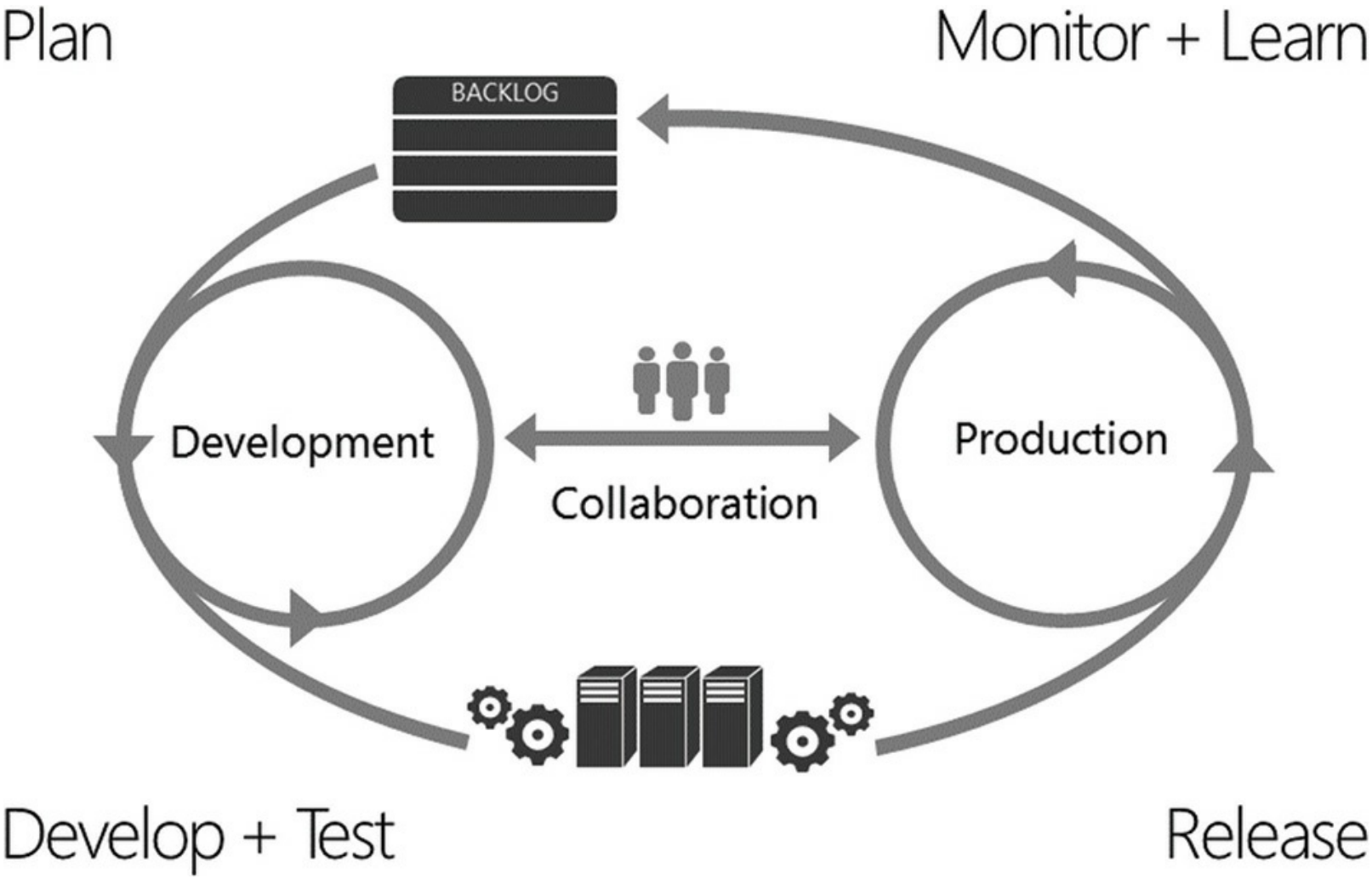

FIGURE 6.1: DevOps workflow from planning to release

- **Plan:** Use a backlog that defines a 1- to 3-week sprint, and define a prioritized list of user

stories. For an ecommerce app, an example of a user story would be that the product catalog microservice enables a customer to browse for items grouped by category.

- **Develop and Test:** Next, develop and test the user story, which includes creating automated unit, performance, and integration tests, which are discussed later in this chapter.
- **Release:** When the new version is ready to release, use an automated process for deployment to make it repeatable and reliable.
- **Monitor + Learn:** Diagnostics and monitoring after a release enables the team to understand how customers use your services and monitor service health.

Whereas Chapter 7, "Monitoring," discusses monitoring and diagnostics in detail, the key here is to remember that you aren't "done" once you deploy your release.

Modern DevOps

Organizations that have fully embraced DevOps are redefining what it means to be agile. Teams are not waiting until the end of the sprint to ship—they are shipping updates to their microservices dozens or hundreds of times a day! While it might seem counterintuitive, high-performing DevOps companies shipped thirty times more frequently and had fifty percent fewer failures by leveraging automation (Puppet Labs 2014 State of DevOps). The Mean Time to Repair (MTTR), which is the average time taken to repair an issue, was twelve times faster for companies that deploy small, more frequent releases than companies that do large, less frequent releases.

When your organization embraces DevOps, deployment stops being an "event," meaning there are no special meetings required to deploy a new version of an app. It's just something that happens whenever it's needed, using an automated pipeline that is managed and monitored from check-in to production.

To summarize, let's look at what a world with and without DevOps might look like in Table 6.1.

Category	Without DevOps	With DevOps
Quality of Code Check-Ins	Unknown	Validated through unit tests
Environment Creation	Manual	Automated
Deployment Frequency	1 to 2 times a month (or less frequently)	Deploy whenever needed, even several per day
App Deployment Process	Requires meetings and planning	Push-button deployment
Deployment Validation	Manual	Automated
Monitoring	Minimal to none	Health and performance monitoring
Dev and Ops Relationship	Culture of blame	Culture of trust

TABLE 6.1: Comparing Teams With and Without DevOps

DevOps Culture

While the majority of this book is about technologies, the key to success with DevOps is people. All of the technology in the world won't help you be successful if your team doesn't work together. Below are just some DevOps culture principles.

Demystify deployments

Rolling out a new deployment can often be described by the team as "terrifying" or "scary." You need to find a way to ensure the entire team knows how to deploy, and that every member of the team is trusted to contribute code to the production environment. The systems put in place to prevent large-scale mistakes should be good enough to ensure that nobody can break the existing experience. To ingrain this in the culture, startups like Lyft have new employees deploy to production on their first day on the job. If something significant breaks, this leads to the next point, the "no blame" rule.

The "No Blame" Rule

No engineer is perfect, and neither are any existing tools or processes. When something breaks, it should never result in finger-pointing and figuring out who will be the scapegoat. Instead, do a post-mortem of the issue, typically called a Root Cause Analysis (RCA), that will objectively explain what happened, why, and how it can be detected and avoided in the future.

Instrument like you need to support it

DevOps often has developers taking responsibility for the uptime of their systems in production. If

you're going to be on call for a production incident, you want to have your system self-heal so you don't get a call to begin with, and if you do get a call, provide enough information to triage it quickly. Code should be written with this in mind, with thoughtful telemetry, health models, and self-healing as part of the design.

Avoid Tribal Knowledge and Collaborate

Tribal knowledge often occurs in development or operations when there is one member of the team who has sole knowledge about a particular system or process—for example, the one engineer who knows how to deploy a legacy app. To avoid tribal knowledge, as a team you want to embrace a culture of collaboration and knowledge sharing. This can be as simple as starting a team Slack channel, OneNote, or wiki where the team members are responsible for documenting and, most importantly, keeping up-to-date team information, internal processes, checklists, utilities, and more.

Work as a Conveyor Belt

Just like a car factory, small pieces of the product should always be ready to ship to production, going through their own iterations of QA and integration within the larger component scope. Use small, frequent, and atomic check-ins for changes in favor of larger, infrequent code changes.

Automation over Technical Debt

By technical debt, we are referring to a set of tasks or work that are either not completed or are done manually "just this one time" to hit a deadline. Technical debt will always happen, but it's important that automation be prioritized. An example for operations would be a manual backup of a server's log files that needs to be automated and set up to run as a scheduled job.

Continuous Integration, Delivery, and Deployment

Continuous Integration, or CI for short, is the process through which developers automate the building and validation of code to ensure quality check-ins. Every check-in is passed through a number of tests that help pinpoint issues early in the development lifecycle.

Continuous Delivery builds upon Continuous Integration and adds a series of practices that ensure the code is always ready to be deployed to production at the click of a button. If at a moment's notice someone decides that the current integrated branch should become the de facto production version, nobody will question the stability and quality of the codebase when it's deployed to production.

Finally, **Continuous Deployment** ensures that once all the tests are passed and there are no conflicting issues, all changes are *automatically* pushed to a production environment. While you can think of it as the next step after Continuous Delivery, it is also a way to make releases as painless as possible, where developers are comfortable with their changes becoming public with no additional effort, while also not stressing about large-scale disruptions as every piece of code is validated using automation before deployment. Table 6.2 summarizes the three processes.

Name	Summary
Continuous Integration	Frequent, frictionless check-ins that maintain code quality through unit tests.
Continuous Delivery	Ensure code is production-ready by running automated integration, load, or stress tests. Promotion to production is manually started by the DevOps team.
Continuous Deployment	Conceptually the same as continuous delivery except that promotion to production is fully automated.

TABLE 6.2: Differences Between Continuous Integration, Delivery, and Deployment

Figure 6.2 shows how each microservice in the Flak.io project defines its own GitHub repository, continuous integration process, and independent continuous delivery pipeline, enabling each service to ship independently of each other. Docker images are built during the continuous integration process and are pushed into a central, private Docker registry. Doing so ensures production realism in that the Docker image that is running in Dev is the same image running in Production, with only per-environment configuration changes, discussed later in this chapter.

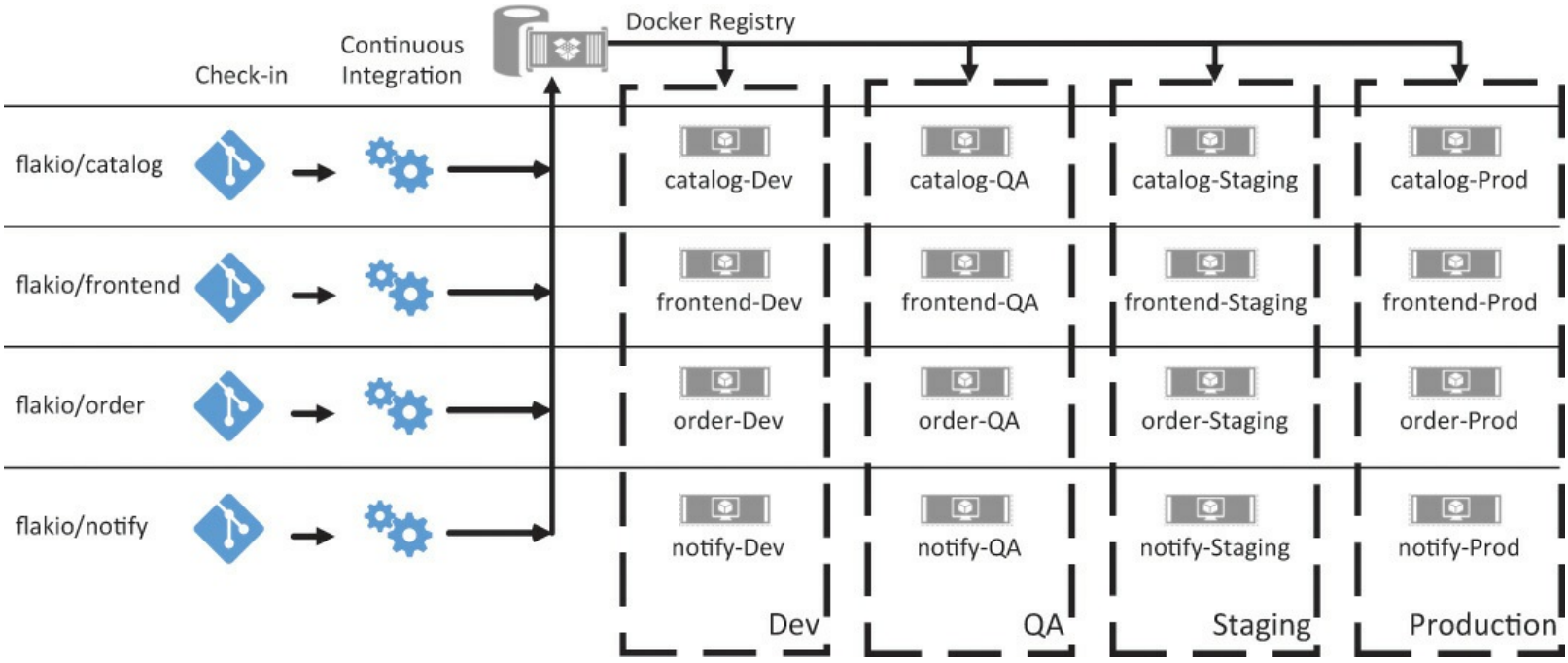

FIGURE 6.2: Each microservice has its own defined continuous delivery pipeline

Creating Environments in Azure

In Figure 6.2, we defined four environments: Dev, QA, Staging, and Production. In this next section, we'll discuss how to use automation and infrastructure to provision these environments in Azure.

Immutable Infrastructure

Before we discuss ways to create our infrastructure, let's discuss two common ways to define your infrastructure and deployments. The first is the classic model, where a server is running for a relatively long time, with patches and updates applied regularly. Any breaking changes result in

tweaks to the server configuration that must be rolled out and tested, resulting in regular maintenance and patching. The longer the server runs and the more deployments that happen against that environment, the higher the risk of encountering issues caused by the state of the machine being different from when it was originally provisioned. One example is a log file that has filled a local hard drive so it now throws exceptions because of a lack of disk space, after weeks of running smoothly. Don't be fooled by server uptime as an indicator of system health—it is not. A system being able to run for an extended period of time is not directly correlated to its inherent stability and ability to properly handle incoming deployments.

Another much more efficient model is defined by the term "Immutable Infrastructure," coined by Chad Fowler of 6Wunderkinder (now part of Microsoft). The term implies that all infrastructure, once deployed, cannot be changed. Therefore, whenever a new configuration or operating system patch is required, it should never be applied to running infrastructure. It must always be a new, reimaged instance, started from scratch. This makes things much more predictable and stable as it eliminates the state factors that might negatively impact future deployments. Immutable Infrastructure heavily relies on the assumption that your environments are fully automated in a repeatable and reliable process, from testing to configuration to deployment and monitoring.

There are three key issues related to the classic model:

- **No or minimal automation:** This means that everything that breaks will require dedicated attention—increasing operational complexity that results in higher maintenance costs that are hard to offset through mutable reconfigurations and updates.
- **Slower and buggier deployments:** The more moving pieces, the more likely that one of them will break and bring the entire system down. It's not uncommon that the perceived modular architecture of the classic model is in fact a monolithic house of cards where an inadvertent change can have damaging consequences.
- **Expensive diagnostics:** Once something actually fails, it will be both time- and resource-consuming to pinpoint the real reason. The fix is rarely a long-term one, but rather a Band-Aid that targets just one potential symptom.

> **Automation doesn't Mean Error-Proof**
>
> Although automation is great, if you aren't testing your automated scripts, bad things can happen consistently and more quickly. Imagine an automation script updating an invalid configuration script to thousands of servers in seconds! It's important to treat automation and scripting as first-class citizens, meaning that scripts are versioned, they use source control, they are tested, and they include instrumentation and tracing for diagnostic purposes.

Infrastructure as Code

Another key aspect of automation is automating the creation of environments. As we discussed in Chapter 5, "Service Orchestration and Connectivity," Azure provides an automated way to create your application topology using Azure Resource Manager (ARM) templates. For DevOps, we'll want to ensure that the creation of all environments is fully automated using ARM templates, including any necessary installation or configuration scripts. Having infrastructure definitions available as code also means any team member can instantly run a script and provision a private instance of your team's

environment painlessly.

Private Versus Shared Environments

There are many approaches you can use to define your infrastructure, but in the next section, we'll look at two examples, a private and shared model. In Figure 6.3 you see a high-level infrastructure view of our continuous delivery pipeline where each microservice defines its own private infrastructure across Dev, QA, Staging, and Production. The benefit of this infrastructure is that each microservice team has complete isolation and control of their infrastructure. The downsides are that there can be a lot of unused or underutilized resources, and a higher cost to maintain this infrastructure.

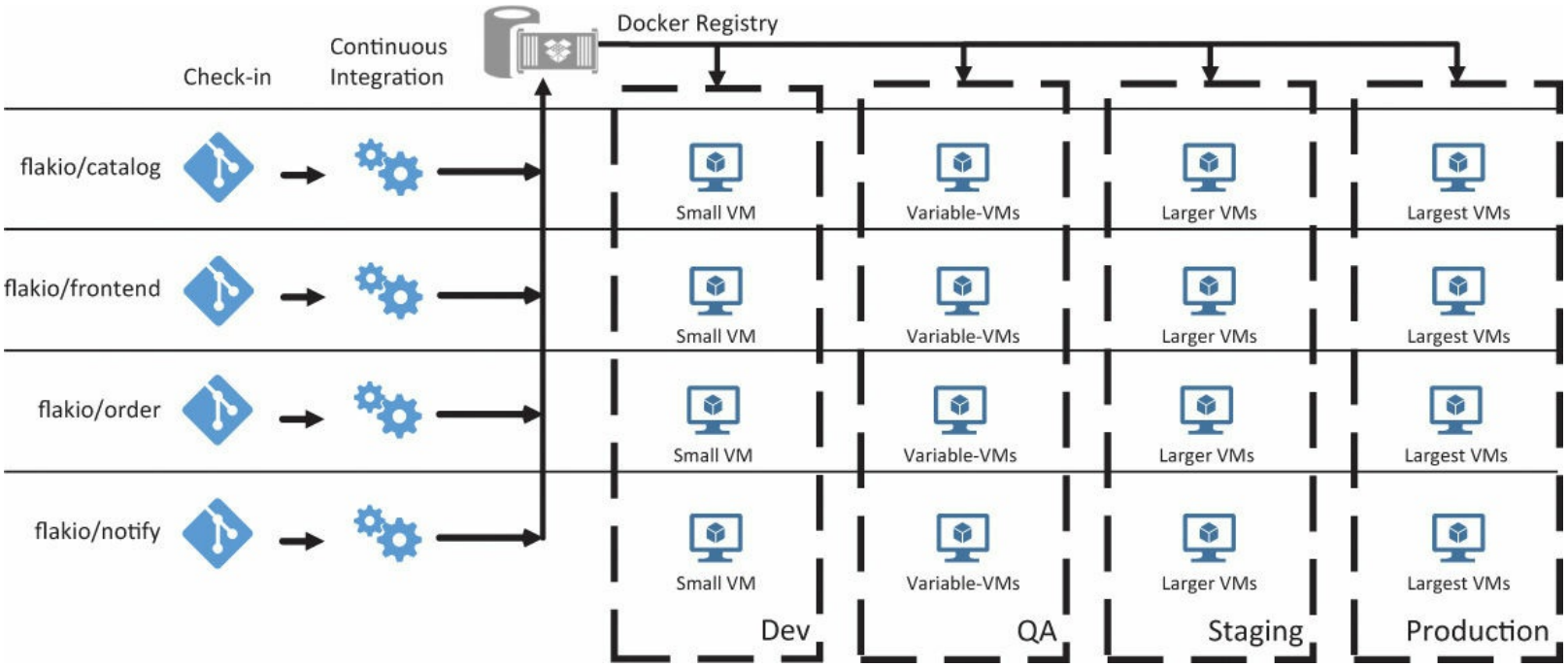

FIGURE 6.3: Each microservice pipeline defines its own private Azure resources

Although Figure 6.3 should be intuitive in that the closer we get to production, the larger and more realistic the preproduction environment becomes, the one thing to note is that the QA environment can be of variable size. This means that, depending on the type of test being done in QA, you can easily scale the number of virtual machines up or down as needed. In other words, you only pay for what you need based on the test needed.

In the semi-shared model shown in Figure 6.4, each team still manages its development environment privately, but all other environments use a shared pool of resources using a clustering technology like Azure Container Service, Docker Swarm, Mesosphere, or Service Fabric as discussed in Chapter 5 under "Orchestration."

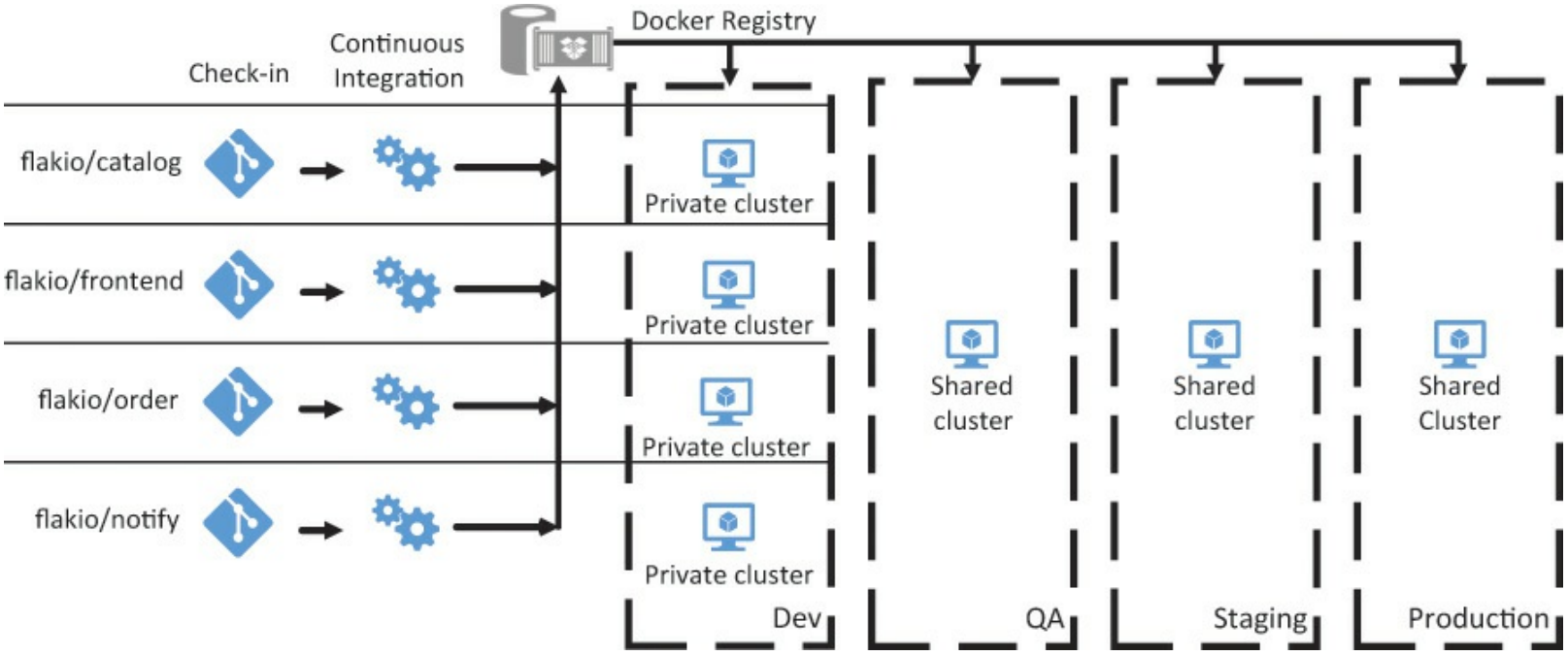

FIGURE 6.4: Microservices using a combination of private and shared resources

Additional Azure Infrastructure

While this is a high-level view of a per-environment topology, each microservice or environment can include a number of other Azure resources that you need to provision. Some examples include

- **Application Insights:** Each microservice and environment should define its own App Insights resources to monitor diagnostics.
- **Key Vault:** An Azure service that stores machine or application secrets.
- **Load Balancer:** As discussed in Orchestration in Chapter 5, this will load balance across Docker hosts.
- **Storage:** Storage provides a durable and redundant data store for containers that use Docker volume mounting.
- **Virtual Network:** You can create custom virtual networks and subnets as well as connections between multiple virtual networks.
- **Virtual Private Network (VPNs):** For microservices that require access to on-premises apps or data stores, you can create a site-to-site connection to directly and securely communicate with your on-premises network.

Creating Environments using Azure Resource Manager

Azure Resource Manager (ARM) enables you to define the desired state of your infrastructure using a set of declarative JSON files. For our continuous delivery pipeline, we can use ARM templates to automate the creation of the four environments shown in Figure 6.3: Dev, QA, Staging, and Production. You can choose to create an ARM template from scratch or start by customizing one of the many prebuilt ARM templates available on GitHub at https://github.com/Azure/azure-quickstart-templates.

Remember from the discussion on ARM, that a common approach when creating multiple environments is to generalize the ARM template, meaning that instead of hard-coding variables into the ARM template, you parameterize the inputs of the template. For a virtual machine template, you

can parameterize the virtual machine size, name, Azure region, or other variable values into a parameter file. The result might look something like this:

- **azuredeploy.json:** The ARM template for your microservice.
- **azuredeploy.*dev*.parameters.json:** Dev environment parameters.
- **azuredeploy.*qa*.parameters.json:** QA environment parameters.
- **azure.deploy.*staging*.parameters.json:** Staging environment parameters.
- **azure.deploy.*production*.parameters.json:** Production environment parameters.

For example, the "Simple Deployment of an Ubuntu VM with Docker template" from https://github.com/Azure/azure-quickstart-templates/tree/master/docker-simple-on-ubuntu includes the following three files:

- **azuredeploy.json:** The ARM template that includes the virtual machine and the Docker virtual machine extension, an extension that automates installing and configuring Docker and Docker Compose.
- **azuredeploy.parameters.json:** The parameterized set of variables that can be different per environment. The exact parameters created in this template are shown.
- **metadata.json:** Documentation or a description file that describes the content of the ARM template.

The azuredeploy.parameters.json file includes parameters for the Azure storage account, location, admin username and password, and DNS name (for example, mydevserver).

Click here to view code image

```
...
"parameters": {
   "newStorageAccountName": {
    "value": "uniqueStorageAccount"
   }
   "location": {
    "value": "West US"
   },
   "adminUsername": {
    "value": "username"
   },
   "adminPassword": {
    "value": "password"
   },
   "dnsNameForPublicIP": {
    "value": "uniqueDNS"
   }
}
```

One common per-environment configuration setting missing in this parameter file is the capability to configure VM size. When creating environments, you probably have smaller VM sizes in development, like an A1 Basic VM with 1 core and 1.75GB of RAM, but your production environment would have a more powerful VM size configuration like the D4 size that includes 8 cores, 28GB RAM, and a 400GB SSD drive. We can parameterize the VM size by defining it as a new parameter in the azuredeploy.parameters.json file as shown here:

```
   "vmSize": {
```

```
        "value": "Standard_D4"
      },
```

Next, you will need to open the azuredeploy.json file and under the **hardwareProfile** property, change the **vmSize** property to read the value from the newly added vmSize parameter.

Click here to view code image

```
"properties": {
 "hardwareProfile": {
    "vmSize": "[parameters('vmSize')]"
},
...
```

In this simple configuration, you can use one ARM template to define the virtual machine and have four parameter files that represent the different per-environment configuration settings.

For more advanced configuration scenarios, there are also templates to take and customize for Docker Swarm, Mesosphere or the Azure Container Service used in the examples for this book.

- **Docker Swarm:** https://github.com/Azure/azure-quickstart-templates/tree/master/docker-swarm-cluster
- **Mesos with Swarm and Marathon:** https://github.com/Azure/azure-quickstart-templates/tree/master/mesos-swarm-marathon
- **Azure Container Service** using the example from this book: https://github.com/flakio/infrastructure

All these examples provide parameter files that you can use to change the total number of VMs included in your cluster (represented as the **nodes** parameter in Docker Swarm template, the **agents** parameter in Mesos, and the **agentCount** parameter in the Azure Container Service template).

> **Windows Server 2016 Containers Preview**
>
> At the time of this writing, Windows Server Containers support is still in preview, but an Azure Resource Manager template and the corresponding container configuration PowerShell script (containerConfig.ps1) is available on GitHub at http://bit.ly/armwindowscontainers.

Tracking Deployments with Tags and Labels

In a microservices architecture, you could have thousands of containers running on hundreds of hosts, and even have multiple versions of the same service running at the same time! To help organize and categorize your infrastructure and applications, one key feature to leverage is adding metadata through the use of Resource Manager tags and Docker labels.

Azure Resource Manager includes the capability to set up to 16 arbitrary tags created as key/value pairs that enable you to differentiate between environments (Dev, QA, Staging, and Production), or between geographical locations (Western U.S. versus Eastern U.S.), or between different organizational departments (Finance, Marketing, HR) to better track billing.

To do this, open the dev environment parameter file and set the values for environment and department as shown. The location tag is already included as a parameter so we don't need to add it

again.

```
    "environment": {
        "value": "dev"
      },
      "dept": {
       "value": "finance"
      },
...
```

Next, in the azuredeploy.json file, we will add a tags section to include metadata about the environment, location, and department. Instead of hard-coding these values, notice that the tag values are read from the parameters we created previously.

Click here to view code image

```
    {
      "apiVersion": "2015-05-01-preview",
      "type": "Microsoft.Compute/virtualMachines",
      "name": "[variables('vmName')]",
      "location": "[parameters('location')]",
      "tags": {
          "environment": "[parameters('environment)]"
          "location": "[parameters(location)]"
          "dept": "[parameters(dept)]"
      }
...
```

Doing this enables you to easily find and filter your ARM resources by tag from the Azure portal or the command line. Instead of looking through a list of hundreds of virtual machines, you can filter the list to just the finance department's production VMs.

For Docker, we can use the **label** command to set key/value metadata about the Docker daemon (meaning the host), a Docker image definition, or when a Docker container is created. For example, you can set a tag on the Docker daemon running on the host to specify capabilities such as an SSD drive as shown, using the reverse domain name notation:

Click here to view code image

```
Docker daemon --label io.flak.storage="ssd"
```

Labels on Docker images can be set using the LABEL command in the Dockerfile. Remember that if you add a label to the Dockerfile image, it is hard-coded into the image itself. For that reason, it's best to only add labels in a Docker image for metadata that will not change based on the runtime environment. For per-environment labels, use the "--label" switch in the **Docker run** command as shown:

Click here to view code image

```
Docker run –d \
--label io.flak.environment="dev" \
--label io.flak.dept="finance" \
--label io.flak.location="westus" \
nginx
```

Once you define the labels for your Docker containers, you can use standard Docker commands to filter based on specific label values. This example will show only those running containers that are in

the “dev” environment.

Click here to view code image

```
Docker ps --filter "label=io.flak.environment=dev"
```

Third-Party Configuration and Deployment Tools

Azure supports a number of third-party configuration and deployment tools that your organization might already be using. These include popular configuration and deployment tools like Puppet, Chef, Octopus Deploy, and others. Azure’s virtual machine extension framework includes extensions to install and configure Puppet, Chef, Octopus Deploy, and other software on virtual machines either through the Azure portal or by defining the extension in an Azure Resource Manager template. You can find the full list of supported virtual machine extensions at http://bit.ly/azureextensions.

Dockerizing your DevOps Infrastructure

The growth of Docker has also started a new trend, to “Dockerize” everything. To Dockerize something is to create a Dockerfile for applications or infrastructure so that it can be built as a Docker image and deployed as a Docker container. There are a number of tools used in DevOps, including source control, continuous integration, build servers, and test servers. Many of the applications you would have once spent hours to download, set up and configure are now distributed as preconfigured Docker images. Here are just some examples:

- For source control, there are many Git Docker images available including GitLab’s community edition: https://hub.docker.com/r/gitlab/gitlab-ce/
- For continuous integration, Jenkins CI is available as a Docker image: https://hub.docker.com/_/jenkins/
- For improving code quality, you can use the SonarQube Docker image to run static analysis tests to discover potential issues, ensure coding style guidelines are met, and get reports for code coverage and unit test results: https://hub.docker.com/_/sonarqube/
- For build, unit, or integration tests, you can use a Docker container as an isolated host for compiling your project, running unit tests, or running coded-UI tests using tools like Selenium: https://hub.docker.com/r/selenium/
- Applications and services like Maven, Tomcat, RabbitMQ, NGINX, Cassandra, Redis, MySQL, and others are all available as Docker images from http://hub.docker.com
- Even the Azure command-line interface (CLI) is available as an image: https://hub.docker.com/r/microsoft/azure-cli/

Deploying a Microservice with Continuous Delivery

In this section we’ll discuss how code flows from check-in to production in a continuous delivery pipeline as shown in Figure 6.5.

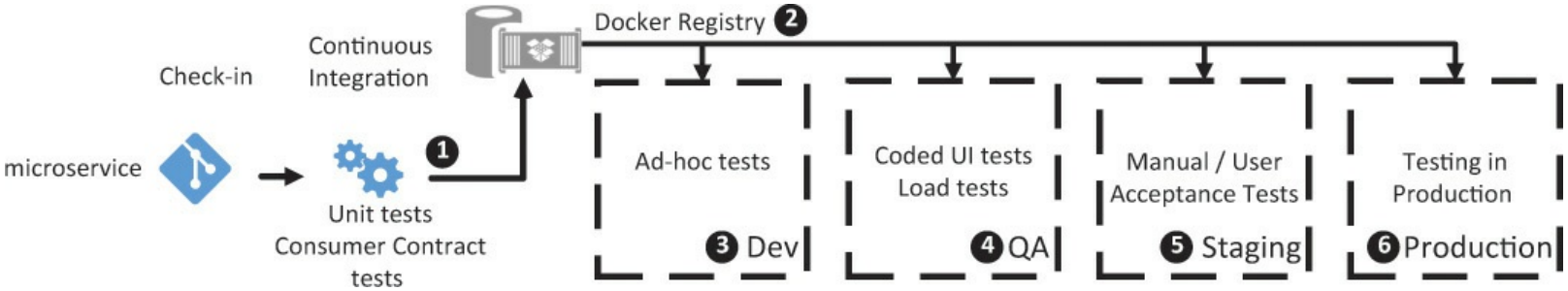

FIGURE 6.5: A microservices continuous delivery pipeline from check-in to production

1. Continuous Integration is when a developer checks in code, a build is automatically triggered, and a set of unit and consumer contract tests (discussed later) are run against the build. If the tests pass, we execute a **docker build** command to build an image and set the version number for the new service. Once the image is built, it is pushed to a Docker Registry (see Chapter 4, "Setting Up Your Development Environment," for Docker Registry hosting options).
2. Create containers in different environments. One of the most important differences between traditional deployments and Docker deployments is that once the Docker Image is built during the CI step, you never again access or build source code in other environments. Instead, you create instances of containers based on the image that was pushed into the Docker Registry during the CI process. Doing this enables teams to get closer to production realism, where the same exact image that was tested in development is run in production. To create a container in an environment, you have a few options. You can call **docker run** to create an image from your Docker repository, or you can use multi-container configuration files like Marathon or Docker Compose to create your container and any interdependent containers it requires.
3. With the microservice now packaged as a Docker image, the Dev environment pulls the set of Docker images that make up the microservice so that developers who need to see a "Dev" version of the microservice have a sandbox for developing and testing their service.
4. Like the Dev environment, the QA environment also pulls and creates Docker containers to run automated tests like integration tests, coded-UI tests, or load tests to further validate the quality of the build. If the tests pass the QA environment, this triggers a deployment to the staging environment.
5. In the staging environment, the set of Docker images that make up the microservice are again pulled from the Docker Registry and we are now ready for production. At this stage, there could be a set of manual tests or user acceptance tests to validate that the service is working well against a more production-like environment. As the dotted line between staging and production shows, the promotion of the microservice from staging to production is manual for continuous delivery.
6. Once the team is ready to promote to production, the production environment again pulls from the Docker Registry to create the Docker containers needed to run in production and a new set of tests are run in production to further validate the quality of the deployment.

Application Configuration Changes Across Different Environments

In the last section, we created containers based on the same Docker image in each environment. A microservice will have configuration settings that likely need to change per environment, like feature flags to dynamically control service capabilities (turn on or off product recommendations), API keys

for third-party services, or connection strings to dependent services like a web front end, database, or a logging service. These configuration settings can be changed per environment in a number of ways:

- Use a Docker volume-mounted configuration file (for example, myservice.conf) that is deployed along with the container and is unique to each environment, similar to the example in chapter 2.
- Use environment variables to set per-environment configuration settings, as discussed in chapter 4.
- Use tools like Puppet and Chef, which both include ways to set up per-environment configuration settings.
- Use a dynamic configuration service like Apache Zookeeper, or HashiCorp's Consul, which provide APIs to store and retrieve configuration information. Consul provides both an API to retrieve configuration and a DNS lookup service. The DNS service enables you to have a set URI to something like a logging server at http://teamlog, which can stay the same across environments, but the DNS resolution service would resolve the URI to the appropriate address per environment, for example, http://dev.server01/teamlog.
- For secrets, like passwords or API keys, you can use Azure's Key Vault service or HashiCorp's Vault service to store and retrieve secrets.

Let's now take a deeper look at each stage in the continuous delivery process.

Continuous Integration

The continuous integration process starts when a developer checks in code or merges code from a feature branch into the main branch. This triggers a series of automated tests, most commonly unit tests, to validate the quality of the code being checked in.

Improving Quality Through Pull Request Validation

In distributed version control systems like Git, a request to have a code change accepted into the mainline branch is called a pull request (PR). Common PR submission errors or check-in rules can be first validated using an automated analysis tool or "bot" that is triggered during a check-in. One example of a bot is rultor (https://github.com/rultor): a bot that, once triggered, will create a Docker container, merge and pull the request into the master branch, run the predefined range of tests, and at the end of a successful run will close the request. Post-checkup, rultor can be used to start a new Docker container and perform the deployment of the existing product to the production endpoint.

Another example is CLAbot (https://github.com/clabot), a bot that will check a pull request that is submitted in a repository to see if the PR author has already signed a Contributor License Agreement (CLA). The CLAbot is so useful that the Azure team uses it to automate the work of validating licensing for contributions to Azure GitHub repositories (https://github.com/azurecla).

Continuous integration defines a build workflow that includes all the steps required to build and run tests for your application. A typical build workflow could contain steps like the following:

- Download source code and any package dependencies like Maven for Java, NPM for Node, or

NuGet for .NET applications.

- Build the code using build tools like Ant, Gradle, or MSBuild.
- Run a set of tasks using Javascript task runners like Grunt or Gulp to optimize images, or bundle and minify JavaScript and CSS.
- Run unit tests using tools like Junit for Java, Mocha for Node, or xUnit for .NET applications.
- Run static analysis tools like SonarQube to analyze your source and code coverage reports, or run specialized tools like PageSpeed for web performance.
- If the tests were successful, push the new image into your Docker registry.

Now that we've discussed what a CI workflow might look like, let's discuss some of the testing and analysis tools mentioned previously.

Unit Testing

A unit test is designed to test code on a functional level. Let's take an example of a simple `Add()` method that takes two numbers and returns the sum. A unit test could run a number of tests to ensure the method worked (1 + 1 = 2), and didn't work (0 + "cat" throws an exception) as expected. One of the main premises of unit testing is code isolation, where a function can be tested independently of any other moving parts. Unit tests help with regression testing, where a code change didn't inadvertently break the expected behavior of an existing test.

Testing Service Dependencies with Consumer-Driven Contract Testing

One of the drawbacks of microservices is that the complexity of running multiples services, each with its own set of dependencies, makes it difficult to test. This can get even more complicated as each service evolves at a different pace from the rest of the system, or worse yet, a potential dependency on an older version of a component that might no longer be available, or whose behavior has changed in a way that could introduce subtle bugs. Going further, a microservice can both be a consumer of, and a provider to, other services. This means you have to clearly articulate microservice dependencies.

One of the ways to define dependencies across microservices is to create *consumer-driven contracts*. As we discussed previously, a good microservice architecture should account for independent evolution of interconnected services. To ensure that this is the case, each one shares a contract rather than a bounded type. A consumer contract is a way to set a list of **expectations** from the service provider (the microservice) that need to be fulfilled to be successfully used. It's important to note that these contracts are implementation-agnostic; they just enforce the policy that if the service can be consumed by clients, the integration tests pass. A sample provider contract for the Product Catalog service is shown in Figure 6.6, where both the FrontEnd and Recommendation services need product information from the Product Catalog service.

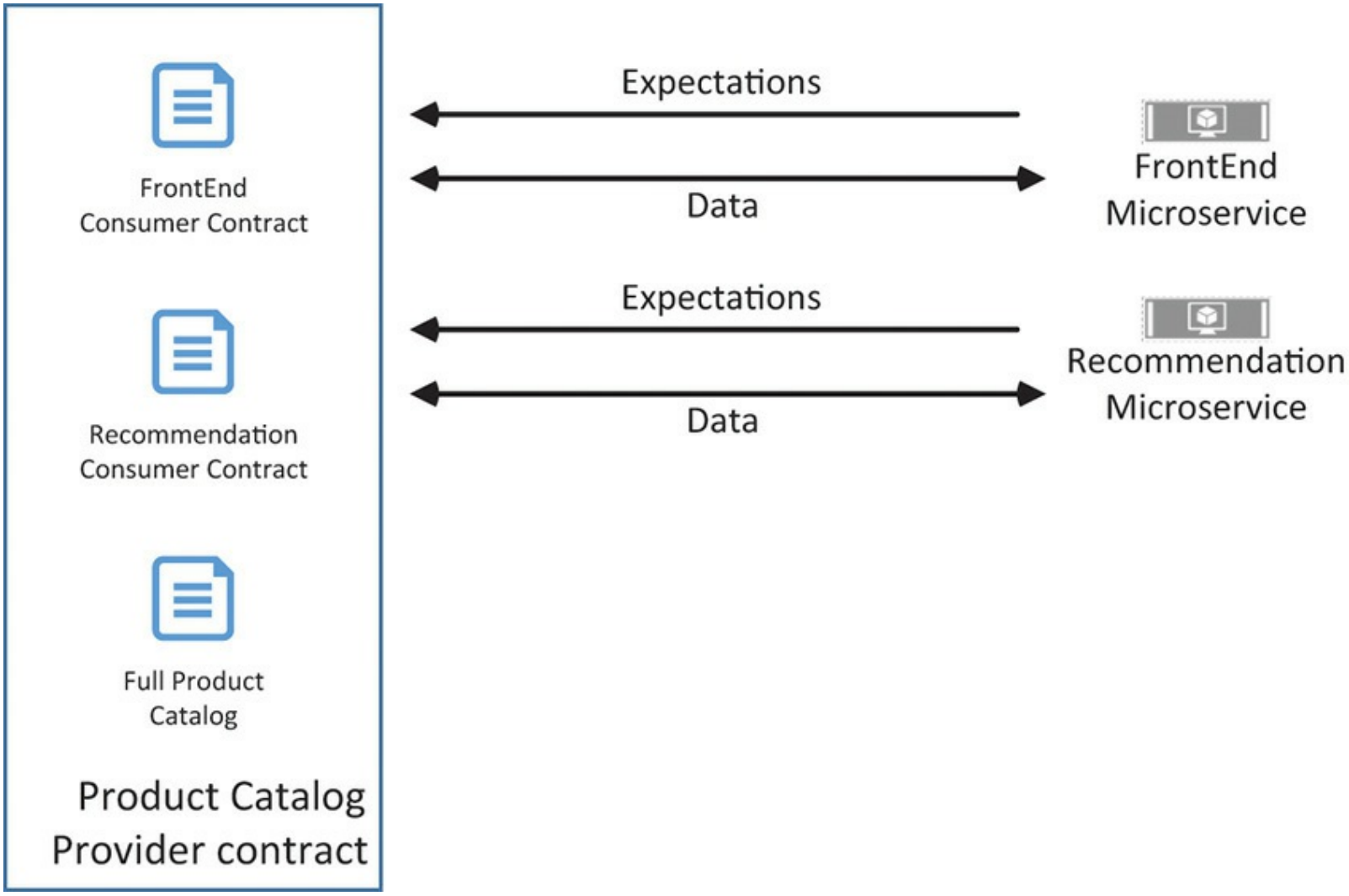

FIGURE 6.6: The provider contract includes two consumer contracts and the full Product Catalog

Ian Robinson outlined three core characteristics for provider contracts in his "Consumer Driven Contracts" overview, from http://martinfowler.com/articles/consumerDrivenContracts.html:

- **Closed and complete:** A provider contract has to represent the complete set of functionality offered by the service.
- **Singular and authoritative:** Provider contracts cover all system capabilities that can be exposed to the client, meaning the contract is the source of truth for what the provider service can do.
- **Bounded stability and immutability:** A provider contract is stable and won't change for a bounded period and/or locale.

In Figure 6.7, both consumer microservices are connecting to the same Product Catalog provider, but with slightly different expectations as to what data they will be handling. While the provider contract exposes the full schema for a provider, the FrontEnd and Recommendation consumers have a different set of required data fields (expectations). The presence of additional fields in the full Provider Catalog should not impact any of the consumers as their only dependency should be on their required fields.

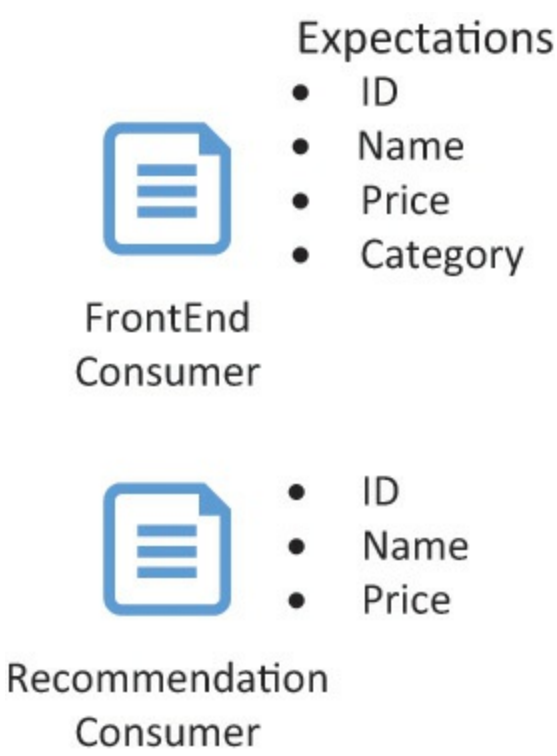

FIGURE 6.7: The FrontEnd and Recommendation consumers each have different data expectations from the Product Catalog provider

In terms of integration testing, client contracts should be explicitly expressed through a range of automated tests that will ensure that no breaking changes are inadvertently introduced into the system. To validate these sets of contracts, a common and flexible approach is to rely on mock objects which "mock" the behavior of a real object or resource, like a microservices REST API. Teams can create mocks that will simulate expected behaviors of services to ensure the service works as expected. Testing your service using mock objects for every check-in ensures that any breaking issues are caught early.

Pact is a consumer-driven contract-testing tool with mocking support that is available for a number of programming languages including Java, .NET, JavaScript, Ruby, and Python, available at https://github.com/realestate-com-au/pact. Pact also includes Pact Broker, a repository for sharing consumer-driven contracts that can even be run as a Docker image, available at https://hub.docker.com/r/dius/pact_broker/.

> **Public and Third-Party Services**
>
> Consumer-driven contracts help ensure that known service dependencies across microservices are well defined and tested both by the provider and consumer. The harder integration testing happens when you are providing a public API or consuming a public third-party API that you don't control. For those scenarios, any changes to an API can break dependent services.

Code Analysis with SonarQube

As code for your service will be added by a number of developers, it is often necessary to ensure that developers follow certain style or quality standards established within your organization. SonarQube is an open-source platform design to continuously monitor code quality including code duplication, language styles, code coverage, documentation comment coverage and more. SonarQube supports a

number of programming languages including Java, JavaScript, C#, PHP, Objective-C and C++.

SonarQube integrates well with a variety of CI tools, such as Bamboo, Jenkins, and Visual Studio Team Services, exposing a web frontend where any engineer can quickly assess the status of the codebase.

Web site performance

Some microservices aren't REST services per se, but rather web sites. For web code, performance is a key metric not only for search engine optimization, but as an Aberdeen group study shows, even a 1-second delay in performance can cause a seven percent drop in conversions. Treat performance like a feature, and ensure that you are measuring your required performance targets. To do this, you can use a number of prebuilt Grunt or Gulp tasks that automate the measurement of your web site performance. For example, you can use the phantomas Grunt task (http://bit.ly/grunt-phantomas), which is a configurable web performance metrics collector with over 100 different built-in performance metrics including image size, caching, browser performance, and more. Performance results are then output to a shared directory, as either a comma-separated values (CSV) file, or using JSON. Another useful tool is the PageSpeed task, which uses Google's Page Speed Insights API to test your site performance for both mobile and desktop, as shown in Figure 6.8. Each performance recommendation includes a link for more information to fix the issue. By integrating speed measurement tools into your CI process, you'll ensure you catch performance degradation bugs with every check-in.

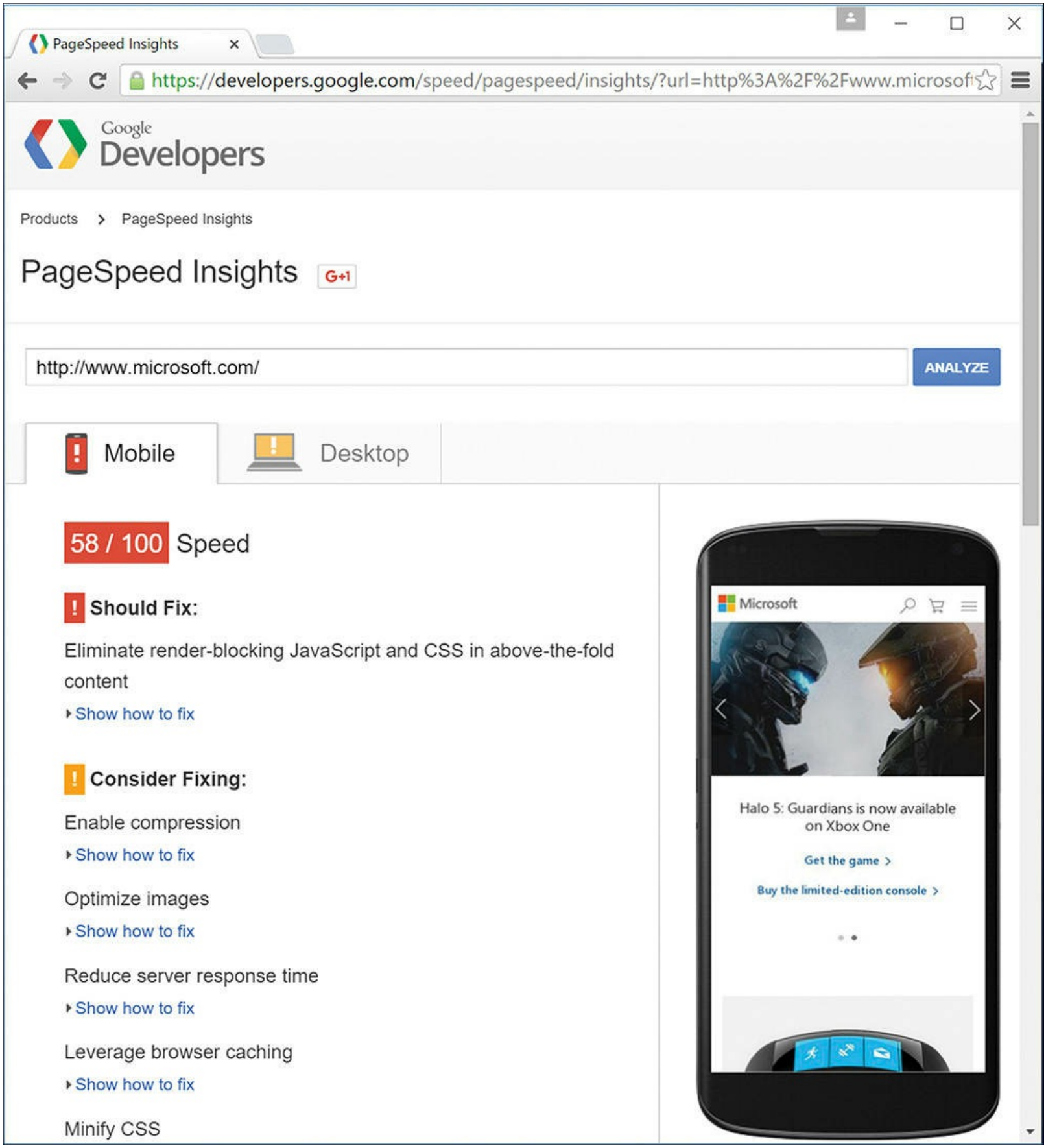

FIGURE 6.8: A sample set of recommendations from Google's PageSpeed Insights web site

Testing in a QA Environment

DevOps is fundamentally about ensuring high quality code. At any point in time your code, which is being pushed from a developer machine into source control, is tested against a number of criteria including scalability, interoperability, performance, and others. Because microservices are commonly

derived from a monolithic app that has been broken down into a set of independent services, it is important to make sure to test and validate that services interoperate with each other, and none are in a state that can break the larger system.

Integration Testing

When you have several microservices as part of a larger project, it is important to make sure that these services work given the interdependence with each other. There are several techniques that can be leveraged for integration testing. For example, Big Bang Integration Testing tests all the services running together at the same time to ensure that the entire system works as expected. Another option is bottom-up integration testing, where services at lower hierarchy levels are exposed to a set of tests, and subsequent services that depend on those services are then tested.

Coded UI Testing

Coded UI testing helps validate that end-to-end scenarios using multiple services work together. For an ecommerce site, this would include searching for a product, adding it to a shopping cart, and checking out to place an order. Coded UI tests help ensure that the integrated multiservice scenario doesn't break.

Selenium (http://docs.seleniumhq.org) is an example of a cross-platform open-source web UI testing framework, that integrates directly with Docker (http://bit.ly/dockerselenium). Selenium tests can either be written in a programming language like Java, C#, or Ruby, or you can use the Selenium IDE which records everything you do in your browser, like the ecommerce browse-to-checkout example. One of the great benefits of Selenium is that it is both independent of the original language in which the web app was written and can run simultaneously in the context of multiple browsers on different systems. This results in a significant reduction of time investment for testing the same project in different situations.

Load & Stress Testing

Among the different types of performance tests, load testing ensures that application performance doesn't degrade based on the size of the load placed on the service. Although a service can have great performance with a hundred customers an hour, you want to ensure the service will still perform under a heavier load when the number of customers increases to 10,000. Stress testing takes this further and tests what happens when you overload a service and it breaks.

Load Testing with Azure and Visual Studio

Microsoft Visual Studio 2015 Enterprise enables developers to record and automate load tests that execute in Azure, and it comes with 20,000 minutes of free usage to get started! These tests can be used for any web site or technology (Go, Java, Node, .NET, and others) and can be customized in a number of ways:

- Set variable time intervals between tests and steps in a test.
- Run multiple tests in parallel to simulate real-world usage patterns.
- Test network latency or mobile performance using network performance values that simulate a 3G or dial-up 56K connection.
- Test multiple browsers, including Internet Explorer, Chrome, and Firefox.

- Geo-distribute your load tests by having them run in different Azure regions like Western U.S., Northern Europe, or East Asia as shown in Figure 6.9.

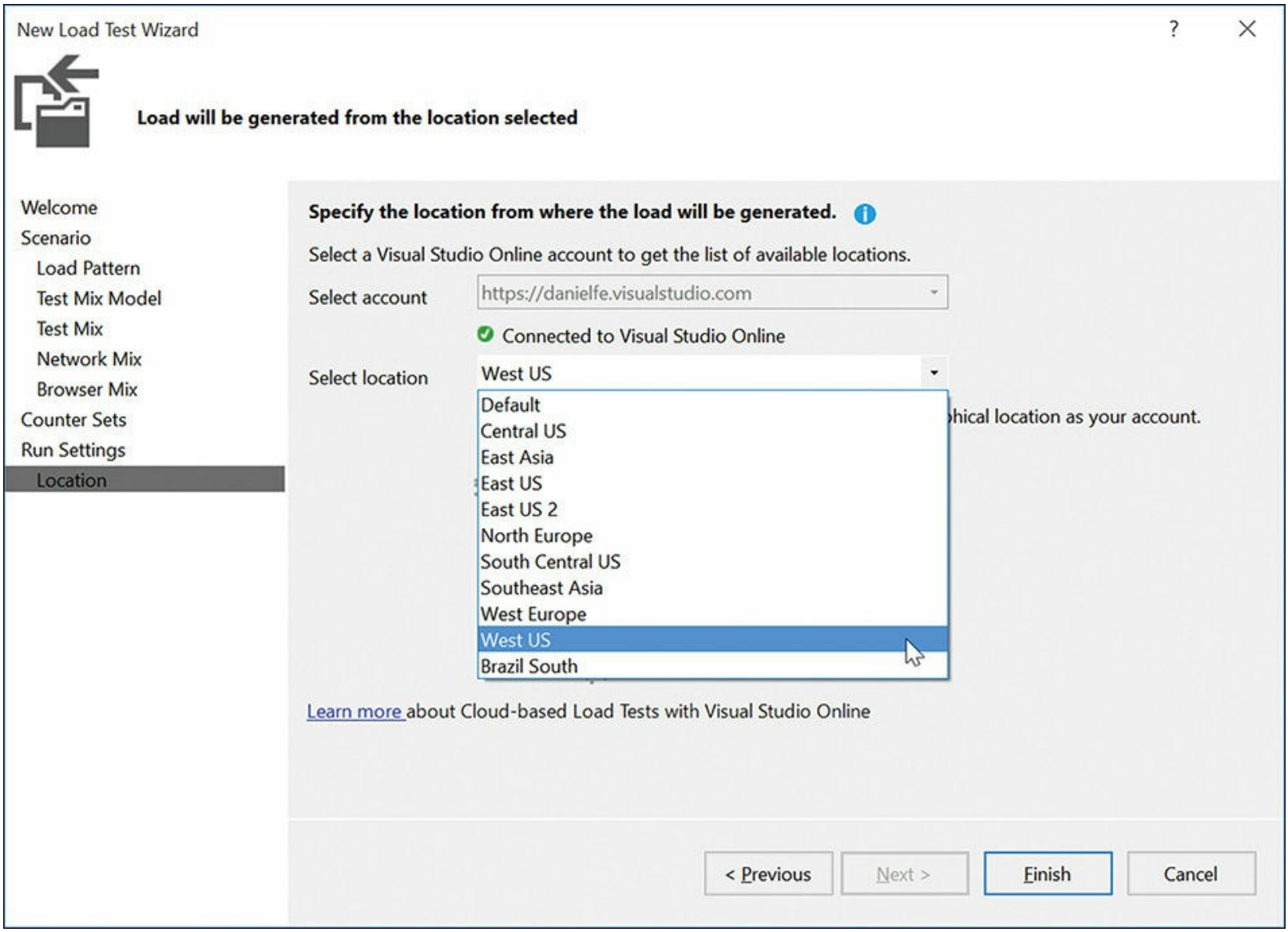

FIGURE 6.9: Selecting the Azure region to run for a load test

By running the test you can start seeing real-time results as shown in Figure 6.10. You can find more information on how to configure and customize load tests at http://bit.ly/azureloadtest. Because these tests are scripted, they can also be integrated directly into your continuous delivery pipeline.

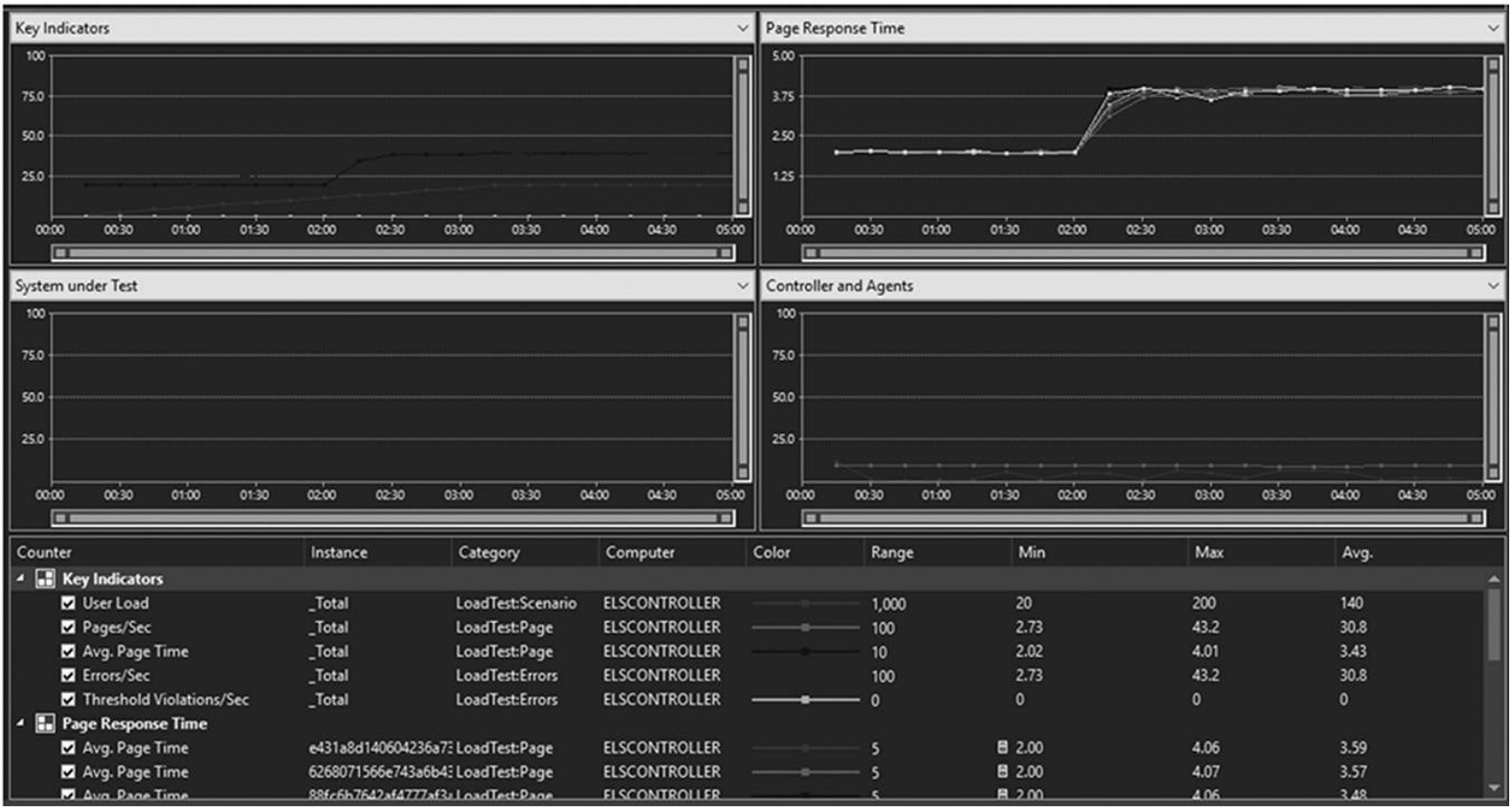

FIGURE 6.10: Viewing the results of a load test

Docker Stress Testing

When working with Docker containers, it is often useful to be able to launch multiple containers on a single host—think of a distributed deployment scenario. Spotify, a popular music streaming service, deploys Docker containers across their server farms, both in testing and production environments. Given their scale, it is important to know when certain clusters might fail, so Spotify decided to create their own tool to handle stress testing—docker-stress available at https://github.com/spotify/docker-stress. Docker-stress enables Spotify to test the container density for a Docker host, meaning how many total containers it can have running at the same time. The tool can spawn a number of worker containers within a single host that can be alive for a limited period of time. Running it can be done through the following command:

```
./docker-stress -c 100 -t 15
```

This code will create 100 containers with a lifespan of fifteen seconds each. Docker-stress also enables you to monitor the status of the stress test using docker-monitor. The docker-monitor command shown performs a health check over a fixed time interval (500 seconds) and sends any failure results to the specified email address.

Click here to view code image

```
./docker-monitor –t 500 –e den@contoso.com
```

Deploying to Staging

A staging environment is the deployment target where the release candidate resides. It is considered the "ready-to-go" code that needs to go through any final validation processes, like user acceptance testing. User acceptance testing is where a user, such as a business owner for a process, validates that

the service is working correctly. There can be a number of additional differences to test related to the environment configuration as well, such as the target databases and services that the application is connected to. Whereas in the development and QA environments the database and service hooks point to independent test endpoints, a staging environment can point to your production database.

Manual/Exploratory Testing

This is one of the most primitive test types, but at the same time it can produce first-hand results on what the end user will go through, and is useful for your development team to quickly validate or reproduce reported bugs. In the case of microservices, manual testing might involve trying to see whether a specific REST call can cause unhandled exceptions and impact the overall system on a larger scale. However, in DevOps manual testing is rarely used, if at all—it is inefficient in a dynamic environment and prone to missed cases. What can be done manually can often be easily automated.

Manual Testing as a Service

If your organization wants or even requires manual testing—for example, a financial services company that requires a security or compliance audit before a release—one option is to use a manual testing service to do this in as close to an automated way as possible. Rainforest QA (http://bit.ly/rainforestqa) provides "human testing at the speed of automation." When you promote your app, from QA to staging for example, or from staging to production, RainforestQA can listen to a webhook in your CI system and start a manual test run performed by real live people.

Testing in Production

As we mentioned in the beginning of this chapter, the process of deploying an application to production can be downright frightening. But there are a number of ways you can use to avoid the risk and fear of pushing a new version out to production. Even when your application is fully rolled out, you're still not done testing, as there are a number of other tests you can do to ensure you have resilient and reliable services.

Canary Testing

Canary testing is a technique used to deploy a new version of your microservice to a small percentage of your user base to ensure there aren't any bugs or bottlenecks in the new service. To do canary testing, you can use tools like NGINX's **split_clients** module to split traffic based on your routing rules. Another option is Netflix's Zuul (http://bit.ly/netflixzuul), which is an Edge service that can be used for canary testing new services based on a set of routing rules. Facebook uses Gatekeeper, a tool that gives developers the capability to only deploy their changes to a specific user group, region, or demographic segment. Similarly, at Microsoft, the Bing team has a routing rule where all employees on the corporate network get "dogfood" experimental versions of the Bing search engine before it is shipped to customers.

For an example, let's say we had a new version of a microservice to canary test. You can split traffic as shown in Figure 6.11, with 99% of initial traffic going to the current release (v1.0), and 1% of traffic would go to the new release (v1.1). If the new release is performing well (see Chapter 7, "Monitoring," for more information on what to measure), then you can increase the amount of traffic incrementally until 100% of traffic is using the new release. If at any point during the rollout, the new

release is failing, you can easily toggle all traffic to use v1.0 without needing to do a rollback and redeployment.

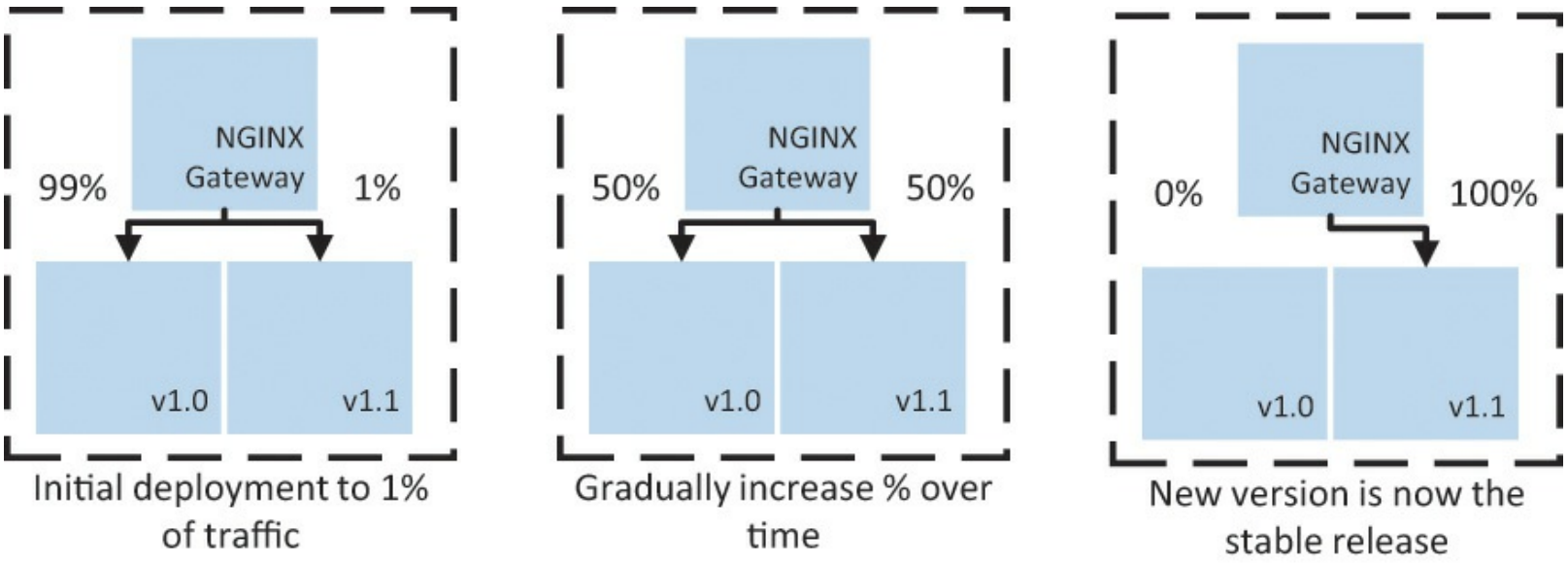

FIGURE 6.11: Deploying a new microservice version using canary testing

Depending on how complex your deployment is, you can do canary testing to roll out across machines and geographies. For example, if the rollout of a logging service for a virtual machine works on one machine, you'd slowly roll it out to all machines within a service, and then to all services within a region. Once that region is successfully up and running, your rollout continues to the next region until the service is fully deployed. Don't confuse canary testing with A/B testing. Canary testing relates to the concept of deploying a build to a subset of your users to ensure quality, whereas A/B relates to feature changes to see if they increase a key performance metric.

A/B Testing

A/B testing is a way to measure how two versions of a feature stack up against each other in terms of performance, discoverability, and usability. When setting up an A/B test, the test manager sets up two groups of users: One is the control group, and the other is the treatment group, which is normally significantly smaller than the control group. As an example, in Outlook.com, we created an experiment where we moved the built-in Unsubscribe bar from the bottom of an email message body to the top, driven by the hypothesis that users simply were not aware of the "Unsubscribe" feature in the product. We rolled out a new treatment that moved the unsubscribe feature to a more discoverable location in the UI, and rolled it out to ten percent of worldwide Outlook.com users. Over a fixed period of time we analyzed the data that showed a higher usage of the Unsubscribe feature that had been obtained simply by moving it to a more discoverable location.

Fault Tolerance and Resiliency Testing

Fault tolerance, often referred to as fault injection testing, implies that the set of microservices that make up your app has unavoidable and potentially undetectable issues that can be triggered at some point in time. The developer makes the assumption that under certain circumstances, the system will end up in a failing state. Fault-tolerance testing covers scenarios where one or many of the critical microservices fail and are no longer available. The resolution comes down to two choices: exception processing or fault treatment. Exception processing means that the underlying "engine" realizes that something is amiss and is able to continue working to the point where it can notify the user about an

error, though it is itself in an error recovery state, and potentially treat the issue on its own.

On the other hand, fault treatment enables the system to prevent the error altogether and ensures that it can take an action before bad things happen.

As with any other aspect of service building, resiliency does not necessarily follow a linear implementation scale. Just because you need to ensure that your services work in a fault-tolerant environment does not mean that you have to embrace every tool and pattern to do so. In the example of Shopify, an ecommerce company, there is a resiliency maturity pyramid that describes the levels of investments depending on the infrastructure in place as shown in Figure 6.12.

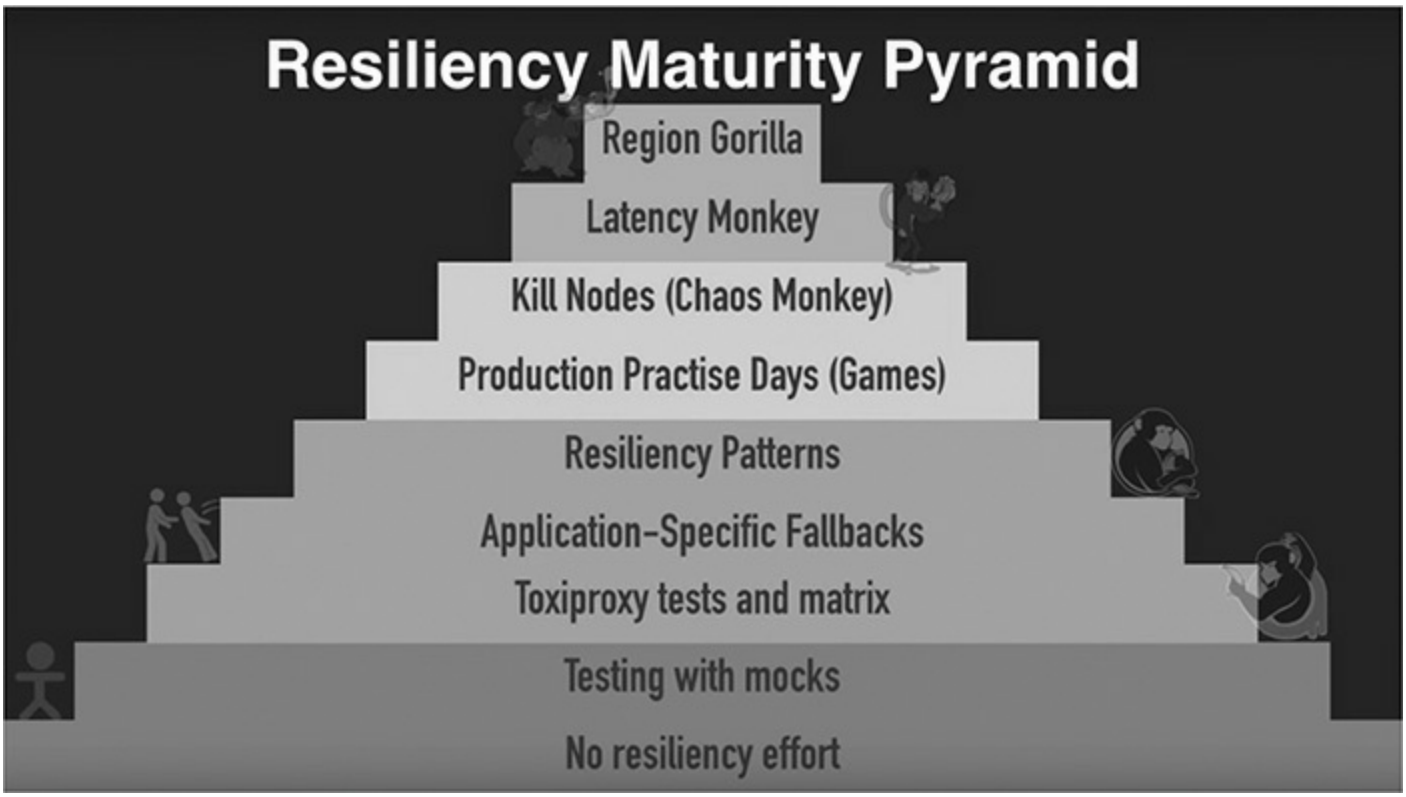

FIGURE 6.12: Resiliency maturity pyramid, presented by Shopify at Dockercon 2015

Moving up the pyramid, for a broader infrastructure it is necessary to test and harden your infrastructure to ensure high availability. Services are required to handle scenarios where components need to be either temporarily automatically decommisioned, or swapped out in a timely manner. Although we discussed testing with mocks earlier, two tools to call out in particular are Toxiproxy and Chaos Monkey:

- **Toxiproxy** is a network proxy tool designed to test network conditions and validate against single points of failure caused by the network, available at https://github.com/Shopify/toxiproxy.
- **Chaos Monkey** is a tool that is part of the Symian Army resiliency tool set, that identifies groups of systems (Auto Scaling Groups, or ASGs) and in a random fashion, picks a virtual machine within that group and terminates it. The service runs on a predefined schedule, in a way that would enable engineers to quickly react to any critical service downtime and test if the system can self-heal against catastrophic failures. The tool is available at http://bit.ly/netflixchaosmonkey.

Choosing a Continuous Delivery Tool

When designing the continuous delivery process, it is critically important to assess the different

options available. Although there is no one-size-fits-all solution, this section outlines some of the key questions you'll need to ask, and compares how some different services stack up against each other. This is not a comprehensive guide for everything that is out there, but it should give you a good idea of key players in the DevOps world and how you can leverage those in your organization.

On-Premises or Hosted?

The first option to consider is whether you want to use an on-premises continuous delivery pipeline or use a hosted service. For on-premises, the organization becomes the maintenance and service provider for the tools and infrastructure, including hardware, networking, upgrades to newer versions, and support issues. Organizations that typically use on-premises tools do so either because of an existing investment in on-premises tools, or because of a corporate policy that forbids source code hosting outside of the company. Beyond compliance, there are benefits to having a local solution —one of them being the capability to fully control and customize the continuous delivery pipeline as needed, whether that includes integration into custom tools, authentication mechanisms, or existing workflows.

Hosted infrastructure is the alternative to on-premises deployment. Infrastructure is hosted on external servers, usually optimized for scale and geographical region distribution. The host is responsible for all maintenance and management, usually with guarantees of availability and performance through a Service Level Agreement (SLA). Updates to the service are managed by the hosting company, but you do lose full control of hosting the service yourself.

While many companies are using on-premises tools today, Software-as-a-Service (SaaS) solutions continue to grow in popularity and richness. International Data Corporation (IDC) predicts that by 2018, almost 28% of the worldwide enterprise application marked will be SaaS-based (http://www.idc.com/getdoc.jsp?containerId=252568).

On-Premises or Hosted Build Agents?

Whether you choose an on-premises or hosted service, there are many times when development teams need full control over build agents. The build process consists of two things: the build controller and the build agent. A build controller is the orchestrator that manages the pool of existing agents, and the build agent handles the actual build process. Build agents are typically hosted on separate machines or in Docker containers, enabling them to scale out based on demand.

A build agent is responsible for getting the source files, instrumenting the workspace, compiling the code, running all required tests, and then producing the final version of the product. Most of the continuous delivery (CD) solutions available on the market today offer a variation of configurable build agents. For example, Visual Studio Team Services (VSTS) enables an organization to manually create agent pools and subsequently provision those with an unlimited number of build agents that can sport a number of declared capabilities. Whenever a build kicks off in VSTS, agent pools are inspected based on the given demands to pick the right agent. TeamCity, another CD service, offers the capability to create custom build agents that are installed and configured separately from the TeamCity server. Atlassian (https://www.atlassian.com) supports both local and remote Bamboo agents, in addition to being able to handle elastic agents—a flavor of remote agents that run in Amazon Elastic Compute Cloud.

Best-of-breed or Integrated Solution?

Another common question when deciding on tooling is whether to choose the best-of-breed solution for each task a team needs, or select a more integrated solution. For many organizations, this choice depends on whether they have very specific needs that only one vendor can fulfill, existing tools that need to be integrated, or how organizational decisions are made. For example, top-down decision-making tends to prefer integrated solutions, and bottom-up decisions tend toward picking best-of-breed tools. With as many options as there are today, it's easy to fill up your "DevOps cart" with a number of tools, deploy them, and then expect everything to work. Consider a scenario where you leverage Atlassian JIRA for work item tracking, TeamCity for continuous integration, and Visual Studio Team Services for release management. There is a higher cost in ensuring that the right endpoints and extensibility between these tools exist for the workflow that you need.

So what are the risks of running a hybrid pipeline? The main one is that integration across toolchains is often not as smooth as one might think. It's not even as much about the existing tools—for most common scenarios all the right capabilities are exposed programmatically. For example, TeamCity developed an add-on that offers deep integration with JIRA that enables much more than a standard one-way street between the continuous integration service and the work item tracking systems. The drawback is that this extension is not free; therefore there is extra cost incurred for integrating systems that you might already have deployed. For other integration scenarios, this is not that much of a problem—for example, TeamCity integrates directly with Visual Studio Team Services to pull the source code and subsequently perform integration operations. The bottom line is that there will be a certain amount of work to be done to ensure that all pieces work together.

As an alternative, you can opt to use an integrated solution that sports all the capabilities in one package. That way, you won't have to worry whether your build or release process is different or relies on potentially incompatible components, as everything is integrated in an already-configured toolchain. Integrated tools are also highly extensible, so in situations where your team needs an additional tool instrumented for the existing workflow, it's often just a matter of creating a configuration file and pointing it to the right binaries.

Does the Tool Provide the Extensibility You Need?

Another key point is making sure that the tool you select is designed for and capable of extensibility. Most out-of-the box solutions are already at a stage where processes, from check-in, to build, to pushing the bits to production, is so well formalized and documented that you will hardly need to do any additional configuration to make sure that it works. That said, an organization rarely relies solely on the developer toolchain to maintain a product. In a world where users are a core contributing force to the direction of the product, several moving pieces are most likely to be in play. In a hypothetical scenario with a startup, team communications are done through Slack, external users are providing a good amount of feedback through UserVoice, while contributors to the open-source segment of the product are opening issues through GitHub. For tasks the team uses Trello, or Jixee, or a Kanban Tool. In any of these cases, you need the flexibility to make sure the tools you already use are integrated, so deployment errors are sent via Slack messages to the operations team, for example.

When choosing the CD platform, extensibility should not be overlooked as it will be an indicator of how viable it will be to use the platform in the long run. As your requirements change, your tools should be able to change and adapt with them. You and your organization do not want to be in a position where you will need to readjust the entire infrastructure and manually try to extend its capabilities because you switched the customer feedback channel.

Below are some key criteria to consider when choosing DevOps tools.

Does the Product or Service Include Tools to Provision and Deploy using Azure and/or Docker?

Microsoft Azure provides a set of services, such as VM provisioning through Azure Resource Manager templates, extensible storage, and API hooks for virtualized resources, tremendously simplifying the DevOps process. It is important to consider whether the tool you select already supports Azure as a first class deployment target to ensure that not only can you deploy, but that you receive rich diagnostic information when the deployment fails. Similarly, what set of tools or services exist for Docker? Are common workflows, like building Docker images, using Docker Compose for deployments, or pushing/pulling images to a Docker registry included? How difficult is it to integrate orchestration tools like Kubernetes or Mesosphere in your deployments?

Does the Tool Include Integration With Popular Testing Tools?

Whatever the project is that you are working on, you need to make sure that it is rigorously tested across all stages, which means that your CD pipeline should be tightly integrated with a range of possible test tools, such as SonarCube, xUnit, Selenium, and many more. Identify what testing tools your team will use and ensure that the tools provide a way to easily integrate them as part of your continuous delivery process.

Does the Tool Include Ways to Manually or Automatically Promote between Environments?

Depending on your organization's process, microservice deployments could be fully automated from check-in to production (continuous deployment), mostly automated where the process is automated up to production but production deployments are manual (continuous delivery), or fully manual where promotion of each environment is done and signed off by a QA team. Whatever tool you select should ideally have the option for all three or be able to switch between manual and automated environment promotion. One caveat to consider for full automation is the scenario where a QA engineer is still validating an existing build while another build is being pushed into the same environment. It would be inconvenient to just swap the bits, thereby invalidating all the previously-done work. That, of course, is just one scenario, and automated approval is just as important—to move code between environments, you need to decide what the right balance of manual versus automated testing makes sense for your services.

Does the Tool Include Ways to Track a Deployment for Auditing?

We are no longer operating in a world where there is a single deployment happening over the course of a week. Instead, tens (and in some cases hundreds) of deployments are happening, all within a single day. At that point, it is important to know what changes are done, by whom, and at what stages of the process. When picking the tools for your CD infrastructure, consider how extensive the audit coverage is. As an example, Octopus Deploy captures changes to deployment processes, variables, events for projects, environments, deployment targets and releases, as well as enables queueing or cancelling deployments depending on the captured status.

Comparison of Jenkins, Team Services, Bamboo, and Tutum

Continuous Integration is a core part of the process. Let's compare some of the top products in the space: Jenkins, Visual Studio Team Services, and Atlassian Bamboo.

From the perspective of establishing a build, all three of these products provide a somewhat similar boilerplate. Bamboo relies on a Build Plan, Jenkins requires creating a Build Project (also called a Build Job), and Visual Studio Team Services relies on Build Definitions.

In Bamboo, a single plan consists of multiple stages. As we discussed earlier, this might involve getting the source code, collecting all artifacts, running all required tests, and any complimentary parts of the process. Each stage is scheduled to be executed in a sequential order. If one of the stages fails, Bamboo will not automatically step over to the next stage. Within a single Bamboo stage there can be multiple Build Jobs that require different capabilities from the build agent. The great thing about Build Jobs is the capability to run those in parallel, where failure no longer impacts any adjacent jobs. In addition, each job is sequenced in Build Tasks.

Jenkins, unlike Bamboo, does not have the granularity of substages within a single plan. A Build Project is similar to a single build plan, with a single build stage, with a single build job, and a number of child tasks. None of these can run in parallel without the help of the Multijob Plugin. One of Jenkins's key strengths is its third-party ecosystem with hundreds of extensions that make it easy to integrate into any Docker workflow.

Visual Studio Team Services offers the capability to define continuous integration steps as a set of customizable, extensible tasks in a build definition. It also includes support for defining multiple environments like Dev, QA, Staging, and Production environments. Figure 6.13 shows the tasks needed to create the Dev environment, including creating an ARM template, creating Docker containers, and running a PowerShell Selenium script.

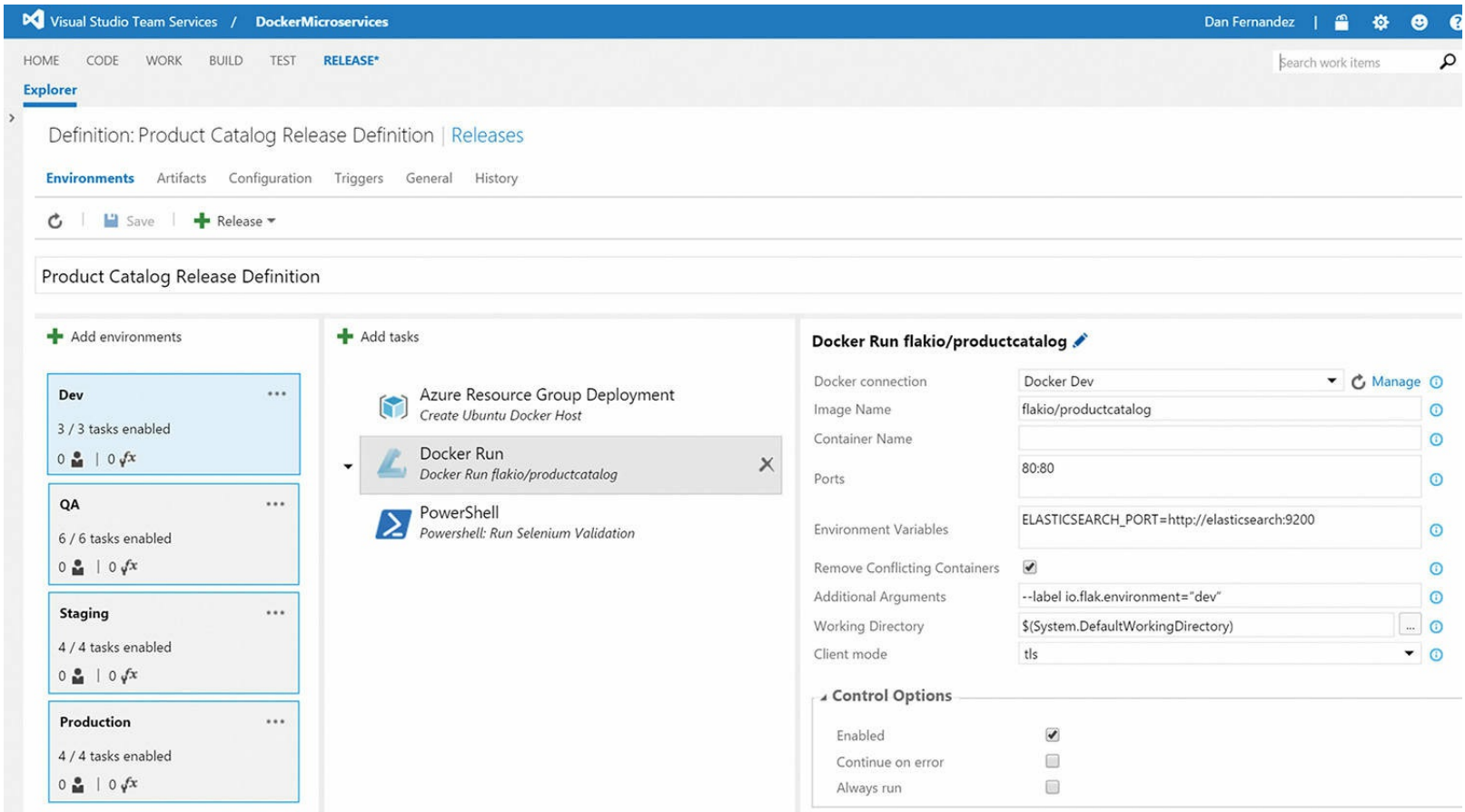

FIGURE 6.13: Team Services environments and tasks

Docker Cloud (Formerly Called Tutum)

Docker Cloud (formerly called Tutum), though still in beta and not recommended to be used in

production, is a platform built with one core goal in mind: to enable you to easily deploy Docker containers from a Docker repository into infrastructure hosted in cloud providers such as AWS, Azure, Digital Ocean, and SoftLayer, as shown in Figure 6.14.

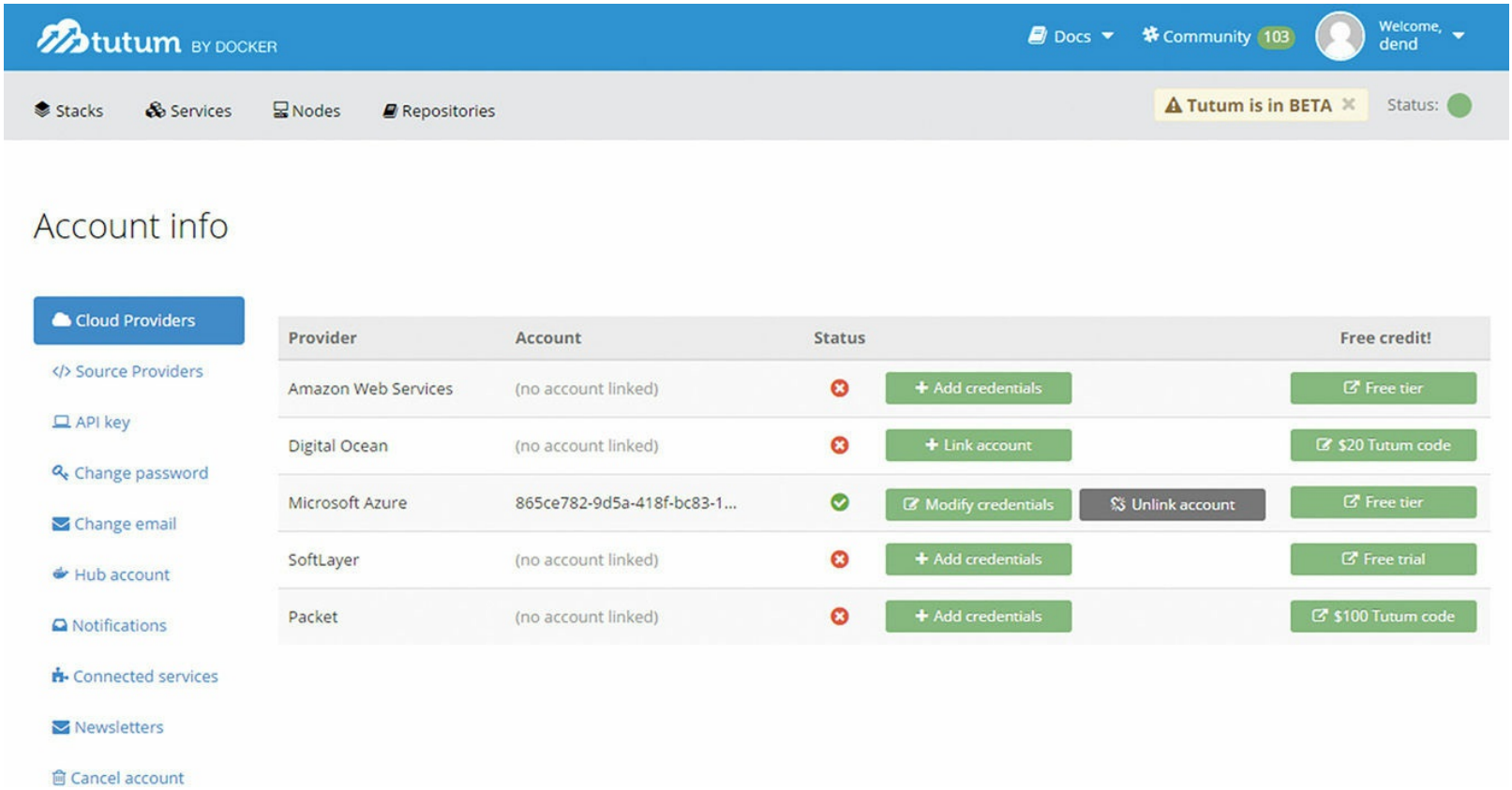

FIGURE 6.14: List of Docker Cloud providers

Docker Cloud also offers a free private Docker registry that will save you the work of setting up your own private registry. In addition, Docker Cloud will ensure that whenever your containers are deployed, the bound node resources are utilized in the most efficient manner for the task, thereby minimizing your bill. While Docker Cloud is arguably one of the most user-friendly tools for deploying containers to the cloud, it doesn't include tools to visualize multiple environments or integrated testing tools to validate your deployment units, coded-UI tests, or performance tests.

The entire Docker Cloud process is composed of five steps:

1. Link the cloud provider to the Docker Cloud account.
2. Deploy a node to the previously connected provider. A node is simply a way to describe a Linux VM that will run your containers.
3. Create a service—a representation of a set of Docker containers belonging to the same repository.
4. Create a collection of services (called a stack) that represent the application as a whole.
5. Manage your repositories that contain Docker images.

Summary

Adopting DevOps practices like automation, continuous delivery, unit testing, integration testing, and performance testing will all have a clear benefit in improving the agility and quality of your releases. While these tools will help make creating a continuous delivery pipeline easier, this process can only succeed based on the people and the collaboration culture you set for your developer and operations

teams.

7. Monitoring

In the last couple of chapters, we have learned how to design and develop a containerized microservice-based application. If we recall some of the core concepts about microservices, we know that the services should communicate with each other through APIs so that we have a loosely coupled and flexible architecture. An environment like this introduces a set of challenges when it comes to monitoring and operational management. Questions like the following arise: how does the system know that it needs to start another instance of a service as one cannot handle the load anymore, or how does the system know that it needs to spin up another host VM as the ones that in use are running into resource constraints? To answer all those questions we need to have effective monitoring in place. In fact monitoring is one of the most important aspects of microservices architectures. In this chapter, we will have a closer look at what monitoring means for each component, what the challenges and best practices for each component are, and we will look at some of the Microsoft monitoring solutions available.

To illustrate that monitoring all components of a typical microservice-based application can be quite a challenge it is worth looking at conceptual view of an environment hosting such an application again. Figure 7.1 shows such an environment that we already know from Chapter 5.

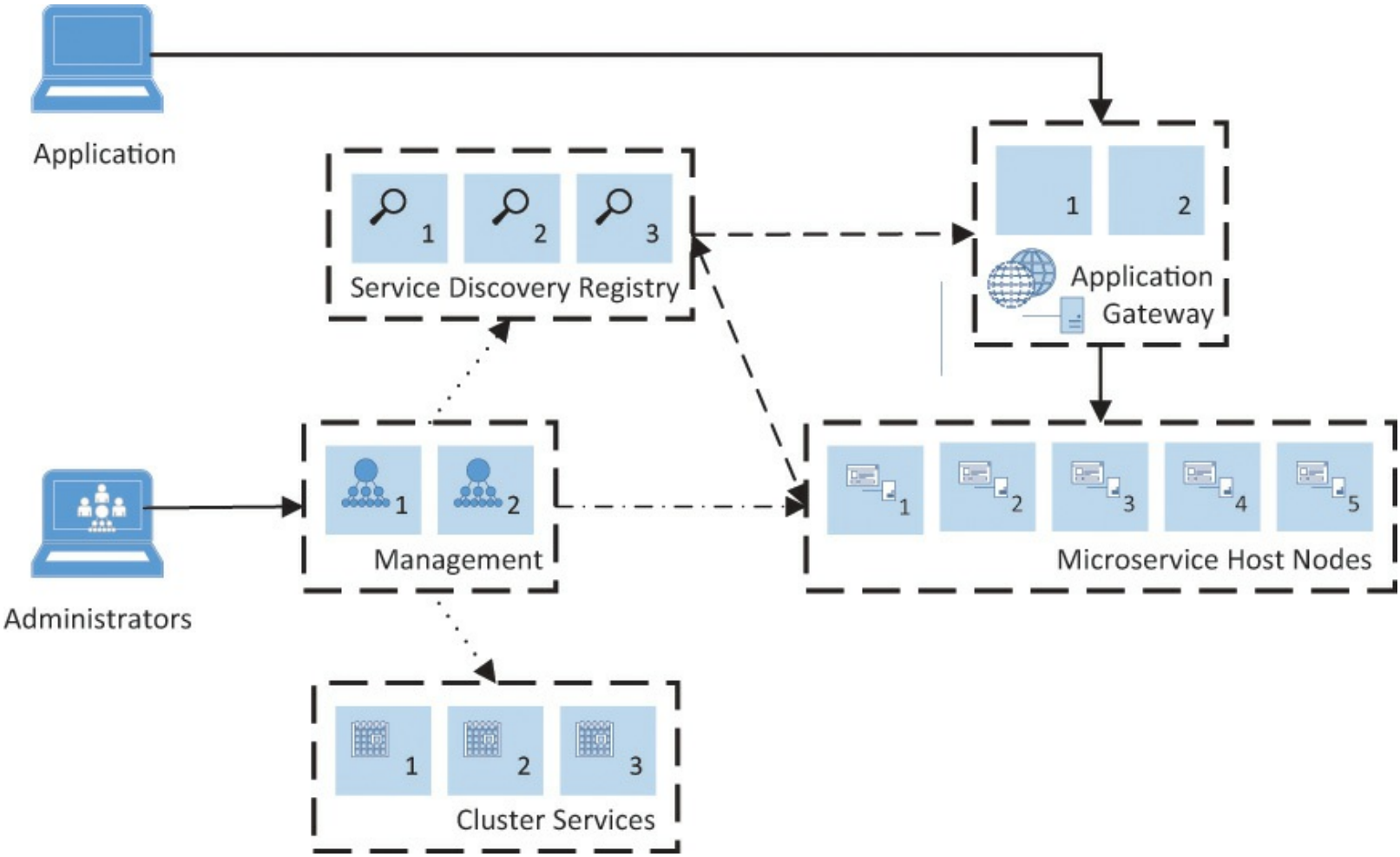

FIGURE 7.1: Microservices-based application

As we learned in Chapter 5, "Service Orchestration and Connectivity," the components of such an environment range from virtual machines that host system services like cluster management or service registries all the way to application services that run in Docker containers. Many of the services are set up in a failover mode, so in the case of a failure of the host VM or service, they can failover to

another host VM. From a monitoring perspective, we can break it down into three big components that need to be monitored:

- The host machine
- The containers
- The services hosted in the containers

Monitoring the Host Machine

What does it mean to monitor the host machine and why do we even need to do that? We have learned that we can spin up containers in seconds, so why not just monitor them?

What happens if for some reason the host machine goes down, as in the case of a scale-in, or it cannot accept requests anymore? Those scenarios do happen, so we need to be ready to respond to them. Another reason for monitoring the host machine is to find out if we are running into resource constraints of any kind. We have learned that we can spin up *n* number of containers quickly, but of course, the number we can really spin up on one host VM depends on resources, such as memory and CPU available on the host system. By monitoring the host machine, we can also find out if we need to spin up a new host machine for new containers, or even if we need to move a container from one host to another. In order to be able to make those decisions we need to monitor the following components at the very least:

- CPU
- RAM
- Network
- Disk
- **CPU:** It is obvious why we need to monitor the CPU of the host system. As we will learn, we can monitor the CPU usage of each container, but we do not have an aggregate CPU usage of all containers. This is a crucial metric to identify whether or not bigger instances of the underlying host are needed or if we should spin up another host instance.
- **RAM:** Similar to CPU, we also want to monitor the aggregate RAM usage of all containers to determine whether we can spin up another container on that host or add another VM to the cluster.
- **Network:** This might not be obvious, but the container endpoints map to endpoints on the host system, so if we have many network applications running, we can find our network interface cards becoming a bottleneck.
- **Disk:** Disk I/O is one of the most neglected metrics we should really be watching. The service performance can highly degrade if many I/O operations are being executed but the disk attached to the host system does not offer high enough IOPs. Containers that contain data storage solutions such as databases are a good example for I/O heavy services.

Monitoring Containers

In a typical containerized environment, one host machine can run many containers. From a macro monitoring perspective, we first want to have an overview of the container status—for example, how many healthy containers vs. broken containers are on that host VM, or how many web containers vs. database containers are running on the host VM? This data is particularly important if we have more

than one VM in a cluster, which is the case in almost any real world scenario. It gives us insights into how and what types of containers and services are distributed across a cluster, so that we can correct unwanted behaviors. For example, we can learn over time that our containers hosting databases use up way more resources than the ones that only serve as gateways. We could then further use the data to tell our scheduler (see more information on orchestration and scheduling in Chapter 5) to only place the database containers on bigger VMs that offer more RAM and CPU, and the gateway containers on smaller VMs.

Therefore, when it comes down to what we should monitor at an individual container level, the classical runtime metrics including CPU, Memory, Network, and Disk are still the important indicators. For example, we can monitor the memory usage trend to detect the potential memory leak caused by a service in the container. Monitoring those metrics on containers is very important as we can combine them with other metrics, such as the ones coming from the host VM. There are situations when only the combined real-time runtime metrics enable us to make the right decisions.

Let's think about the following scenario. We detect that one of our containers has a very high CPU usage. As we have learned, we can spin up a container quickly, so we might think just to spin up another one to add more CPU capacity. However, if the current host environment is running low on CPU, we are unable to put a new container on it. In this case, we would need to add a new host VM first, and then put the container on it.

So how and where does Docker emit the data needed for monitoring? The answer is that Docker relies on two Linux kernel mechanisms, control groups and namespaces (we have discussed control groups and namespaces in Chapter 2), to create the isolated container environment.

Those two features also provide the basic container runtime metrics.

- Control groups expose metrics about CPU, Memory, and Disk usage through a pseudo-filesystem. In most of the latest Linux distributions using the Linux kernel 3.x or later, such as Ubuntu 14.04, CentOS 7 Red Hat Enterprise Linux 7 and so on, they are mounted on "`/sys/fs/cgroup/`" with each control group having its own sub-directory. For example, memory metrics can be found in the "memory" control group sub-directory. On some older systems, it might be mounted on `/cgroup` and the file hierarchies are also different.
- Namespaces expose network metrics. We can utilize the system call *setns* to switch the current monitoring agent process to the same network namespace of the container and then read the metrics data from "`/proc/net/dev`".

Docker Runmetrics

The Docker web site http://docs.docker.com/articles/runmetrics/ provides more information on Docker runtime metrics.

Now that we know where to find the data, we need to have an easy way to read that information. There are actually two choices.

- Read the data directly from the control groups and namespaces
- Use the Docker Remote API

The Docker Remote API is a set of RESTFul APIs using JSON and GET/POST methods. Since Docker Remote API v1.17, it can collect key performance metrics from containers running on the

host. It provides a programmable way for external monitoring agents to query Docker information like container metadata and lifecycle events. The information returned by the APIs provide a comprehensive overview of the containers and their host VM. Below is a list of calls relevant to monitoring.

- **GET /info:** provides system-wide information; for example, total memory of the host, total number containers on the host, total number of images, and so on.
- **GET /version:** provides the Docker version.
- **GET /events:** provides the container lifecycle and runtime events with timestamp. Table 7.1 provides an overview of all events and their respective Docker commands. Monitoring container events is crucial for the overall monitoring strategy in automated environments, as it provides insights into the lifecycle of the containers.

Event	Trigged When	Associated Docker command
Create	Create a new container or run a new container	docker create docker run
Destroy	Remove a container	docker rm
Export	Export a container to a tar archive	docker export
start	Start a newly created container or start a stopped container or restart a running container or restart after the container process exits	docker run docker start docker restart
Restart	Restart a running container	docker restart
Stop	Stop a running container	docker stop
Die	Kill a running container or stop a running container or restart a running container or the container process exits	docker kill docker stop docker restart
Kill	Kill a running container	docker kill
pause	Pause a running container	docker pause
unpause	Unpause a running container	docker unpause
Oom	The container is killed as out-of-memory	
exec_create/ start	Run docker exec command	Docker exec

TABLE 7.1: Overview of container events

As an example, the single API GET /containers/(container_id)/stats can return all CPU, Memory, Disk, and Network usage in the unified JSON format.

Docker API Security

The Docker Remote API is using UNIX sockets, which enables traditional UNIX permission checks to limit access. In order to secure the communication between the Docker client, or any other HTTP client, one should enable TLS authentication. The Docker web site offers more information on how to enable this at https://docs.docker.com/engine/articles/security/

By default, the Docker Remote API is bound to a local UNIX socket (`unix:///var/run/docker.sock`) on the host it is running on. However, it can be bound to a network port on that host so that monitoring agents (and other software that talks to the Docker Remote API) can communicate with it over the network. So potentially, we can configure one monitoring agent to collect container metrics from multiple hosts.

By v1.20, the Docker Remote API only returns all the possible metrics for the given container on one-second interval. There is no way to specify the metrics type and interval. It can generate a significant overhead if we want to monitor hundreds of containers with the Remote API. Thus, if the target environment is resource-restricted, reading the data from control groups and namespaces can be a better choice. On the other hand, Docker is a quickly evolving platform. We also expect the Remote API will provide more custom options in future releases to improve the resource utilization.

> **Note**
>
> The Docker Remote API website https://docs.docker.com/reference/api/docker_remote_api/ offers a complete list of all the different Remote API versions and methods.

Now that we know what we need to monitor from a container perspective, where to find the data, and how to access the data, we need to find a way to collect it. The good thing is that we do not really need to build our own agent to collect the data as there are already many monitoring solutions available, although we could.

An important question, however, is where to run the monitoring agent. Most monitoring solutions offer either a monitoring agent that runs on the host VM, or a container that contains the monitoring agent. While the preferred way is to containerize the agent as well, the answer is really that it depends on the scenario and host VM.

If the VM already hosts other applications, we can extend the monitoring agent running on each host to support Docker as well. It is very doable, irrespective of whether you choose the native Linux solution (control groups and namespaces) or the Docker Remote API. In fact, many existing server monitor solutions have enabled Docker monitoring in their host-based agent. The Azure Diagnostics agent, for example, runs on the host VM, collects all the data from the directories on the host VM, and transfers it to a different location, such as Azure storage.

If our host VMs are only hosting containerized applications, we need a consolidated solution to deploy and manage all the applications, including the monitoring agent. The preferred way is to containerize the agent as well. With the agent and all its dependencies packaged into a single image, we can deploy and run the monitoring agent on any Host/OS and integrate with other container orchestration tools.

> **Monitoring Agents**
>
> Some special Docker environments such as CoreOS, do not even permit third-party packages installed on the host VM, so using a "Monitoring" container is the only option.

Monitoring Services

The last piece in the puzzle is monitoring the services themselves. Having good monitoring in place is important to keeping any application healthy and available. In a microservices architecture good monitoring is even more important. We not only need to be able to monitor what is going on inside a service, but also all the interactions between the services and the operations that span them. When an anomaly occurs, the inter-services information is much needed to understand the causality and find the root cause. It's important to embrace the following design principles to achieve it.

- Log aggregation and analytics
- Use activity or correlation ID's
- Consider an agent as an operations adapter
- Use a common log format

In addition to these points, common standard monitoring tools and techniques should be utilized where appropriate, such as endpoint monitoring and synthetic user monitoring.

Log Aggregation

A request into the system will often span multiple services, and it is important that we are able to easily view metrics and events for a request across all the systems. There is always the question of how much to log to avoid over- or under-logging. A good starting point is to log at least the:

- **Requestor name/ID:** If a user initiates the request, it should be the user name. If a service initiates the request, it should be the service name.
- **Correlation ID:** For more information, see the paragraph on Correlation ID.
- **Service flow:** Log entry and exit points of a service for a given request.
- **Metrics:** Log runtime performance of the service and its methods.

With all the data, can you imagine finding something in the logs of one service, and then having to go to another system and try to find the related logs? For even just a handful of services, this would be painful. This is not something we should spend our time doing, so it should be easy to query and view logs across all the systems.

There are tools that collect and aggregate logs across all the VMs and transfer them to a centralized store. For example, Logstash, an OSS tool by Elastic.co, or the Microsoft Azure diagnostics agent (which we will discuss later in the chapter) can be used to collect logs from all the nodes running our services, and put them in a centralized store. There are also many good tools that help us visualize and analyze the data. One of the more popular end-to-end solutions is the ELK stack. It uses Elastic Search as the data store, Logstash to transfer the logs, and Kibana to view the logs.

Correlation ID

In addition to collecting the logs, we need to be able to correlate logs; basically we need to be able to find associated logs. When a request lands on the application, we can generate an activity or correlation ID that represents that unique request on the application. This ID is then passed to all downstream service calls, and each service includes this ID in its logs. This makes it easy to find logs across all the services and systems used to process the request. If an operation fails, we can trace it back through the systems and services to help identify the source. In Figure 7.2, we can see how a correlation ID can be used to build a waterfall chart of requests to visualize the end-to-end processing time of an operation. This can be used to optimize transactions or identify bottlenecks in

the transactions.

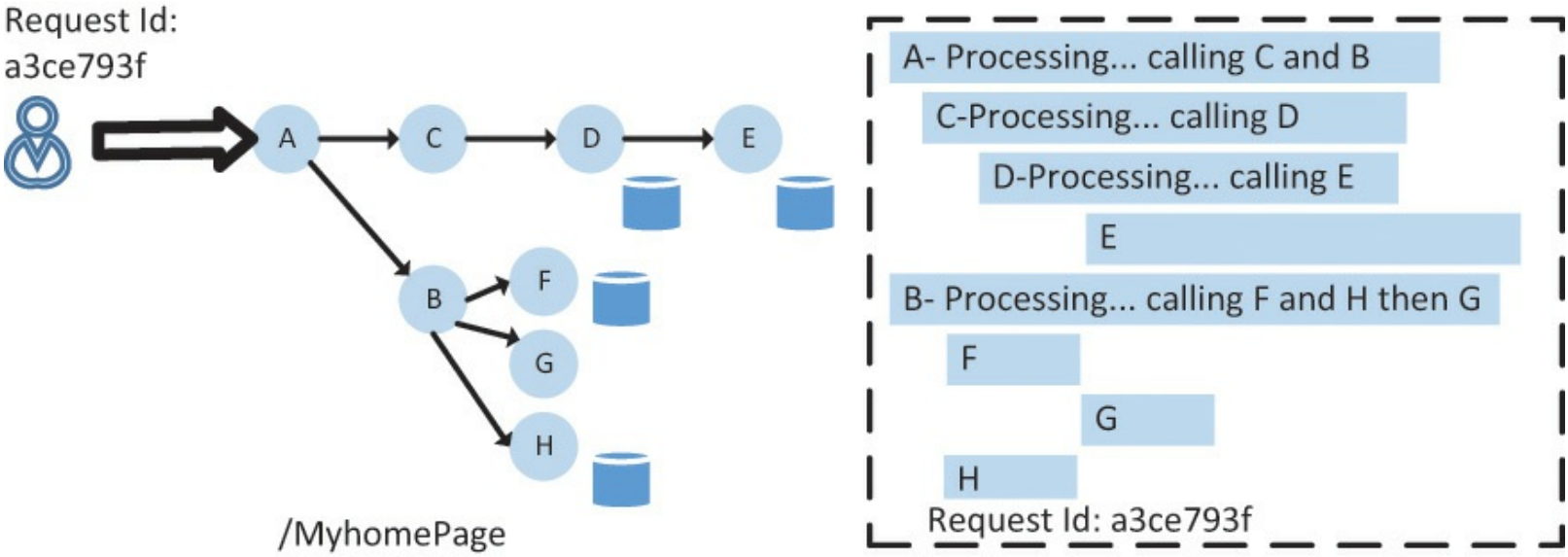

FIGURE 7.2: Correlation ID used in an operation

Operational Consistency

Freedom to use the technology of choice for each service has its benefits, but can present some operational challenges as well. The operations tools and teams now need to deal with a growing number of stacks and data stores, which can make it difficult to provide a common view of system health and monitoring. Every technology tends to deal with configuration, logging, telemetry, and other operational data a bit differently. Consider providing some consistency and standard operational interfaces for things like service registration, logging, and configuration management.

Netflix, for example, uses a project called Prana and a sidecar pattern, also sometimes referred to as a sidekick pattern, to ensure that type of consistency. The Prana service is an operations agent that is deployed to each virtual machine. The operations agent can manage things like configuration in a consistent manner across all the various services. Then the teams implementing the services can integrate with the agent through an adapter and still use whatever technology they want.

> **Note**
>
> The sidecar pattern refers to an application that is deployed alongside a microservice. Generally, the sidecar application is attached to its microservice just as a sidecar would be to its motorcycle, thus the name "sidecar pattern." For more information on sidecar pattern and Netflix Prana visit http://techblog.netflix.com/2014/11/prana-sidecar-for-your-netflix-paas.html.

Common Log Format

Collecting the logs and throwing a correlation ID in the log entry is not enough. We need some kind of consistency in our logs to properly correlate them. Imagine that we write a correlation ID like the following:

Click here to view code image

```
[ERROR] [2345] another message about the event - cid=1
```

Another microservices team implements their logging as follows:

Click here to view code image

```
{type:"info",thread:"2345",activityId:"1",message:"my message"}
```

There are a few things wrong with this. The differing format of the logs is one problem, and then the keys vary across the logs. When we query the logs for cid=1 we are going to miss the logs from the second microservice because although it's logging the correlation ID, they call it something else. The same is true for the event timestamp. If one service logs it as "timestamp", another "eventdate", and yet another "@timestamp", it can become difficult to correlate these events, and time is a common property to correlate events on. Thus, for some of those critical events, we need to make sure that every team is using the same key name, or at least we must consider processing events with something like Logstash.

> **Note**
>
> Logstash is a very popular data pipeline for processing logs and other event data from various systems such as Windows or Linux. Logstash offers many plug-ins for other systems, like Elasticsearch. This makes the log data easily searchable and consumable, making Logstash a great data pipeline for many scenarios.

In addition to the key, the format and meaning of the event message need to be consistent for analysis. For example, timestamp can refer to the time the event was written, or when the event was raised. As timestamp is commonly used to correlate events and analysis, not having consistency could skew things quite a bit.

Further, we need to determine what events should be consistent across the entire organization, and use them across all the services in the company. We should think of this as a schema for our logs that has multiple parts that are defined at different scopes, to facilitate analysis of log events across the various organizational, application, and microservices boundaries.

For example, we might include something like the following in all our log events.

- Timestamp with a key of 'timestamp,' the value in ISO 8601 format, and as close to the time the event happened
- Correlation Identifier with a key of 'activityId' and a unique string
- Activity start and end times, like 'activity.start' and 'activity.end'
- Severity/Level (warning, error) with a key of 'level'
- Event Identifier with a key of 'eventid'
- Process or services identifier enabling us to track the event across services
- Service name to identify the service that logged that event
- Host Identifier with a key of 'nodeId' and a unique string value of the machine name

As services will be developed by multiple teams across an organization, having these agreed upon, documented, shared, and evolved together, is important.

A good example for the importance of enforcing a common log format is Azure itself. Some Azure Services depend on each other. For example, Azure Virtual Machines rely on Azure storage. If there is an issue with Azure storage, it can affect Azure Virtual Machines. It is important for Azure Virtual Machine engineers to be able trace back the issue to its root cause. Azure storage components need to

log the data to the overall Azure diagnostics system following a common format and correlation rules. The Azure Virtual Machine engineer can now easily trace back the issue to Azure storage by just searching for the correlation ID.

Additional Logging Considerations

In addition to the preceding recommendations, we should also consider the following points in our logging strategy:

- **Log what is working:** This can be a simple message to say the service launched successfully, or a regular heartbeat to indicate the service is alive. This might not seem necessary, but it is actually critical for monitoring.
- **Log what is not working and error details:** In addition to the exception message and stack trace, the semantic information including user request, transaction, and so on, will greatly help the initial triage and further root-cause analysis.
- **Log performance data on critical path:** Performance metrics directly reflect how well the services are running. By aggregating the percentile on performance metrics, you can easily find out some system-wide long-tail performance issues.
- **Have a dedicated store for logging data:** If your application uses the same store as your log data, issues with one can affect the other.
- **Log "healthy" state data:** This will enable you to create a baseline of how things look during normal operations.

How to Log Application Data from Within Containers

Now that we have an idea how to structure our logs and what to log, it's time to look at how we can get the log data out of the containers.

For Docker containers, STDOUT and STDERR are the two channels that pump the application logs out of containers. This obviously requires the application to write logs to either STDOUT or STDERR.

Docker v1.6 added a logging driver feature that can send the STDOUT and STDERR message from the container directly to other channels specified in the driver. These include the host's syslog system, Fluentd log system, Journald log system, Graylog log systems, and more.

> **Note**
>
> Find more information on supported logging drivers at http://docs.docker.com/reference/logging/overview.

Monitoring Solutions

By now, we have gained enough knowledge to know how to build our own logging system. However, building custom monitoring solutions is not an easy task, and it only makes sense if there is a requirement that cannot be met by an existing monitoring solution. In fact, most monitoring solutions can be customized in a way to meet almost any requirement. In this section, we will have a closer look at the monitoring solutions offered by Microsoft Azure.

Azure Diagnostics

Azure offers a free basic monitoring and diagnostics framework that collects data from all kinds of components and services on a virtual machine called Azure Diagnostics. Azure Diagnostics is really more like a log data collector than a complete solution, as it does not offer any user interface to view the data. Some monitoring solutions and tools use the data collected from Azure Diagnostics to provide log analysis and alerting and other features.

To get started with Azure Diagnostics, we only need to install the Azure Diagnostics extension on a VM. This can be dynamically applied to a VM at any given time. We can enable the diagnostics extensions through

- CLI tools
- PowerShell
- Azure Resource Manager
- Visual Studio 2013 with Azure SDK 2.5 and higher
- Azure Portal. The diagnostics extension is turned on by default if a VM is created through the portal.

Note

The diagnostics extension for Windows contains the Windows diagnostics agent and the diagnostics extension for Linux contains the Linux diagnostics agent. The Linux diagnostics agent is open source and available on GitHub. https://github.com/Azure/WALinuxAgent.

When we enable the extension through any of the methods mentioned previously, the Linux diagnostics agent is installed on the host VM. As part of the installation, the extension also installs rsyslog, which is important for data logging. Figure 7.3 illustrates the basic components and data flow.

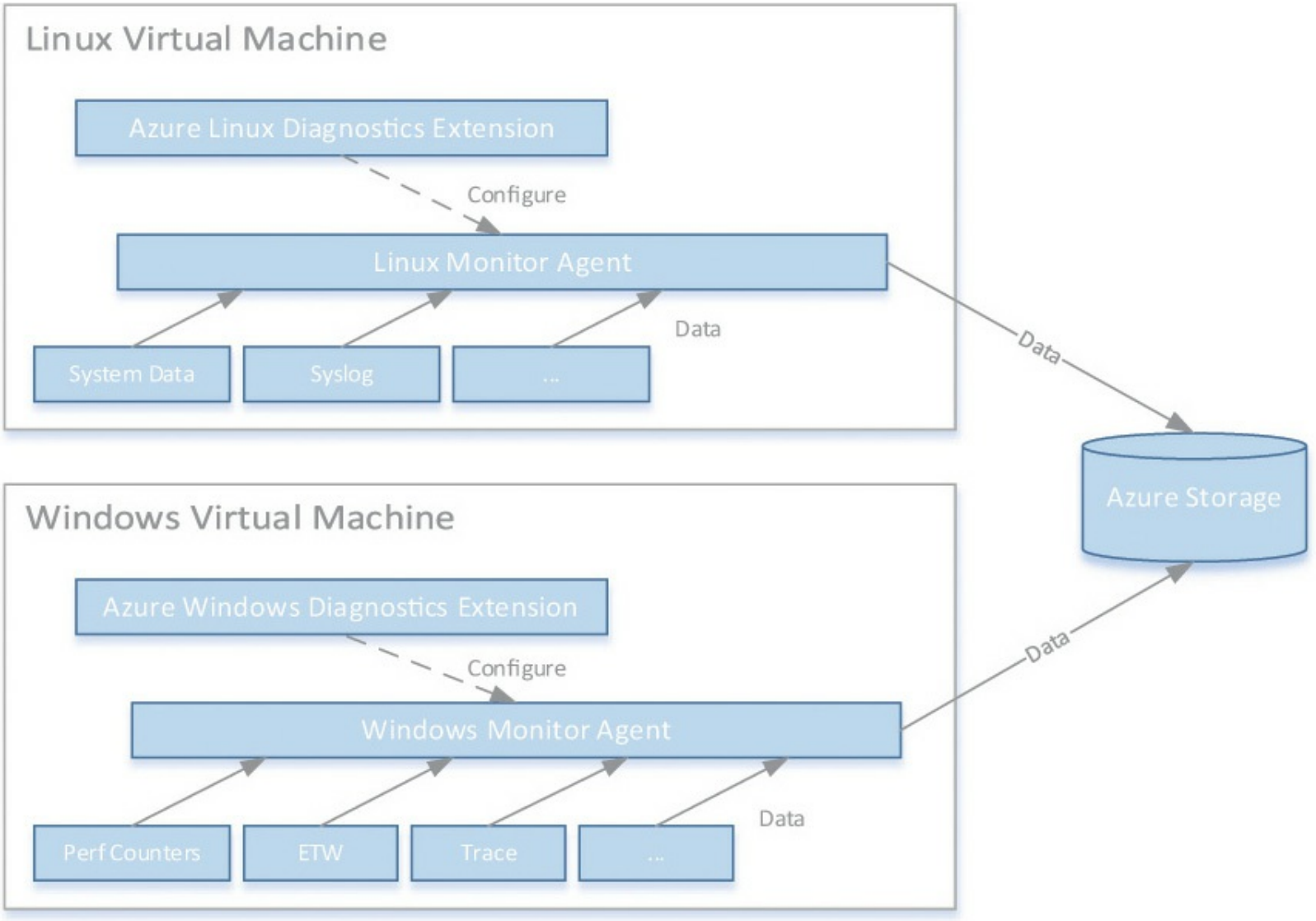

FIGURE 7.3: Linux and Windows diagnostics agent architecture

To enable the Linux diagnostics agent, we need to pass in a configuration that contains a collection plan. The collection plan defines what metrics need to be collected and where to store the data. The data is usually stored in an Azure storage account from which it can be consumed by any client.

As we focus on Docker and Linux in this book, we will not discuss monitoring on Windows. Please see https://msdn.microsoft.com/en-us/library/azure/dn782207.aspx for Windows configuration details.

The following example shows how to collect a core set of basic system data (CPU, Disk, and Memory) and all rsyslog information on a Linux virtual machine.

- Create a file named **PrivateConfig.json** with the following content:

Click here to view code image

```
{
    "storageAccountName":"the name of the Azure storage account
where the data is being persisted",
    "storageAccountKey":"the key of the account"
}
```

- Run the following command from the CLI:

Click here to view code image

```
azure vm extension set vm_name LinuxDiagnostic Microsoft.
```

```
OSTCExtensions 2.* --private-config-path PrivateConfig.json
```

After we have applied the extension, it takes about 5 minutes for the Linux diagnostics agent to be operational. This is very different from a monitoring container, as a container would spin up in almost a matter of seconds and start transferring data.

The Linux diagnostics agent creates the following tables in the storage account specified in the PrivateConfig.json:

LinuxCPU Table's Most Relevant Fields
PreciseTimeStamp: The time when the data is logged in the machine.
PercentIOWaitTime: Percentage of time that the processor spent waiting for IO operations to complete.
PercentIdleTime: Percentage of time during the sample interval that the processor was idle.
PercentProcessorTime: Percentage of time that the processor spent executing a non-idle thread.

Figure 7.4 shows the LinuxCPU table with data.

PartitionKey	RowKey	Timestamp	PreciseTimeStamp	Host	PercentIOWaitTime	PercentIdleTime	PercentProcessorTime
00000000000...	linuxco...	8/31/2015 7:...	8/31/2015 7:27:41 AM	linuxcontainer1	1	99	1
00000000000...	linuxco...	8/31/2015 7:...	8/31/2015 7:28:41 AM	linuxcontainer1	1	99	1
00000000000...	linuxco...	8/31/2015 7:...	8/31/2015 7:29:41 AM	linuxcontainer1	1	99	1

FIGURE 7.4: LinuxCPU table with data

LinuxDisk Table's Most Relevant Fields
AverageReadTime: Average time, in seconds, of a read of data from the disk.
AverageWriteTime: Average time, in seconds, of a write of data to the disk.
ReadBytesPerSecond: Bytes read from disk per second.
WriteBytesPerSecond: Bytes written to disk per second.

Figure 7.5 shows the LinuxDisk table with data.

PartitionKey	RowKey	Timestamp	PreciseTimeStamp	Host	AverageReadTime	AverageWriteTime	ReadBytesPerSecond	WriteBytesPerSecond
000000000...	linuxco...	8/31/2015...	8/31/2015 7:03:41 AM	linuxcontainer1	0.17	0.00738461538461539	170	9147
000000000...	linuxco...	8/31/2015...	8/31/2015 7:04:41 AM	linuxcontainer1	0.17	0.00738461538461539	113	6098
000000000...	linuxco...	8/31/2015...	8/31/2015 7:05:41 AM	linuxcontainer1	0.17	0.00738461538461539	85	4573

FIGURE 7.5: LinuxDisk table with data

LinuxMemory Table's Most Relevant Fields
AvailableMemory: Available physical memory in megabytes.
UsedMemory: Used physical memory in megabytes.
PercentAvailableMemory: Available physical memory in percent.
PercentUsedSwap: Used physical memory in percent.

Figure 7.6 shows the LinuxMemory table with data.

PartitionKey	RowKey	Timestamp	PreciseTimeStam	Host	AvailableMemory	UsedMemory	PercentAvailableMemory	PercentUsedSwap
0000000000...	linuxco...	8/31/2015...	8/31/2015 8:54:...	linuxcontainer1	6810	157	98	0
0000000000...	linuxco...	8/31/2015...	8/31/2015 8:55:...	linuxcontainer1	6810	157	98	0
0000000000...	linuxco...	8/31/2015...	8/31/2015 8:56:...	linuxcontainer1	6809	159	98	0

FIGURE 7.6: LinuxMemory table with data

What about containers? As mentioned before, we can start containers with different log drivers. If we start our container using the syslog-log driver, and the application writes its logs to STDOUT and STDERR, the data is transferred to the LinuxsyslogVer2v0. (The suffix Ver2v0 is the data schema version.)

LinuxsyslogVer2v0 Table's Most Relevant Fields
EventTime: The time when the event occurs.
SendingHost: The name of the host that sends the event.
Facility: The name of the process that creates the event.
Severity: Numeric value that indicates the severity of the event.
Msg: The text content of the event.

Figure 7.7 shows log entries from a microservice inside a container, as well as from the containers itself. The value of the facility field is "daemon" as it refers to the Docker daemon that is logging the message. The Msg field contains the log entry itself, starting with the container ID that looks like "docker/e15bca350018."

PartitionKey	RowKey	Timestamp	Host	EventTime	SendingHost	Facility	Severity	Msg
000000000...	contai...	10/18/201...	containerhost	10/18/2015 3:34...	127.0.0.1	kern	6	[30622.240071] docker0: port 2(veth9c64ef8) entered forwarding state
000000000...	contai...	10/18/201...	containerhost	10/18/2015 3:34...	127.0.0.1	daemon	6	<30>Oct 18 15:34:32 docker/e15bca350018[4369]: 73.225.19.196 - - [18/Oct/2015:15:34:32 +0000] "GET / HTTP/1.1" 302 406 "-"
000000000...	contai...	10/18/201...	containerhost	10/18/2015 3:34...	127.0.0.1	daemon	6	<30>Oct 18 15:34:32 docker/e15bca350018[4369]: 73.225.19.196 - - [18/Oct/2015:15:34:32 +0000] "GET /wp-admin/install.php
000000000...	contai...	10/18/201...	containerhost	10/18/2015 3:34...	127.0.0.1	daemon	6	<30>Oct 18 15:34:40 docker/e15bca350018[4369]: 73.225.19.196 - - [18/Oct/2015:15:34:40 +0000] "POST /wp-admin/install.php

FIGURE 7.7: Log entries from a microservice in a container

Using the Syslog Driver

Start a container using the syslog driver: `$ docker run -d --log-driver=syslog [myimage]`

At the time of writing, the Azure Linux diagnostics agent is certainly a very basic way to collect microservices and container log data. One of the reasons for this is that it stores all log data in one table and does not offer a column to filter based on container ID or name. Another drawback is that Azure table storage might not be the best solution for high-throughput logging requirements due to its indexing limitations. That said, Azure Diagnostics is certainly a good starting point for testing logging strategies and potentially small microservices-based applications. The following Azure website offers more details on the Linux Diagnostics extension and its configuration options: https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-diagnostic-extension/.

Application Insights

Application Insights is a Microsoft Service that collects telemetry information for mobile, browser-based, or server applications. Application Insights collects the data and moves it into the cloud to process and store. The telemetry and diagnostics data can be viewed, as well as sliced and diced, through an integration with the Azure Portal. The following link provides more information on general Application Insights functionality: https://azure.microsoft.com/en-us/services/application-insights/.

Recently, Application Insights added full support for Docker containers by adding an Application Insights image to Docker Hub. Application Insights follows the approach of a "monitoring container" as we had mentioned in the container monitoring section. As a result, we can run a single instance of an Application Insights container on our Docker host VM. The service in the container talks to the Docker agent and sends the telemetry data back to Application Insights.

Application Insights supports two models for Docker.

- Capture telemetry for apps not instrumented with Application Insights. This means there is no instrumentation code in the microservice, resulting in only the following Docker related data being captured:
  - Performance counters with Docker context (Docker host, image, and container)
  - Container events
  - Container error information

This data by itself is already very helpful, as we have learned before. We can get even more out of Application Insights if we instrument the microservice code.

- Capture telemetry for instrumented apps.
  - Gets all the data mentioned before.
  - Adds the Docker context (Docker host, image, and container) to the captured telemetry data.

Note

See https://azure.microsoft.com/en-us/documentation/articles/app-insights-get-started/ for more information on how to instrument applications with Application Insights.

As mentioned previously, Application Insights offers a great UI for looking at and analyzing log data. The experience enables us to drill down into the Docker aspects of our microservices application. Application Insights offers an overview Docker blade with information for:

- Container activity across Docker hosts and images
  - CPU
  - Memory
  - Network in
  - Network out
  - Block I/O
- Activity by Docker host
- Activity by Docker image
- Active images
- Active containers

In Figure 7.8, we can see the Docker overview blade for our book companion application Flak.io, showing that we currently have four Docker host machines in this cluster, the activity of the four Docker images, and the number of active containers.

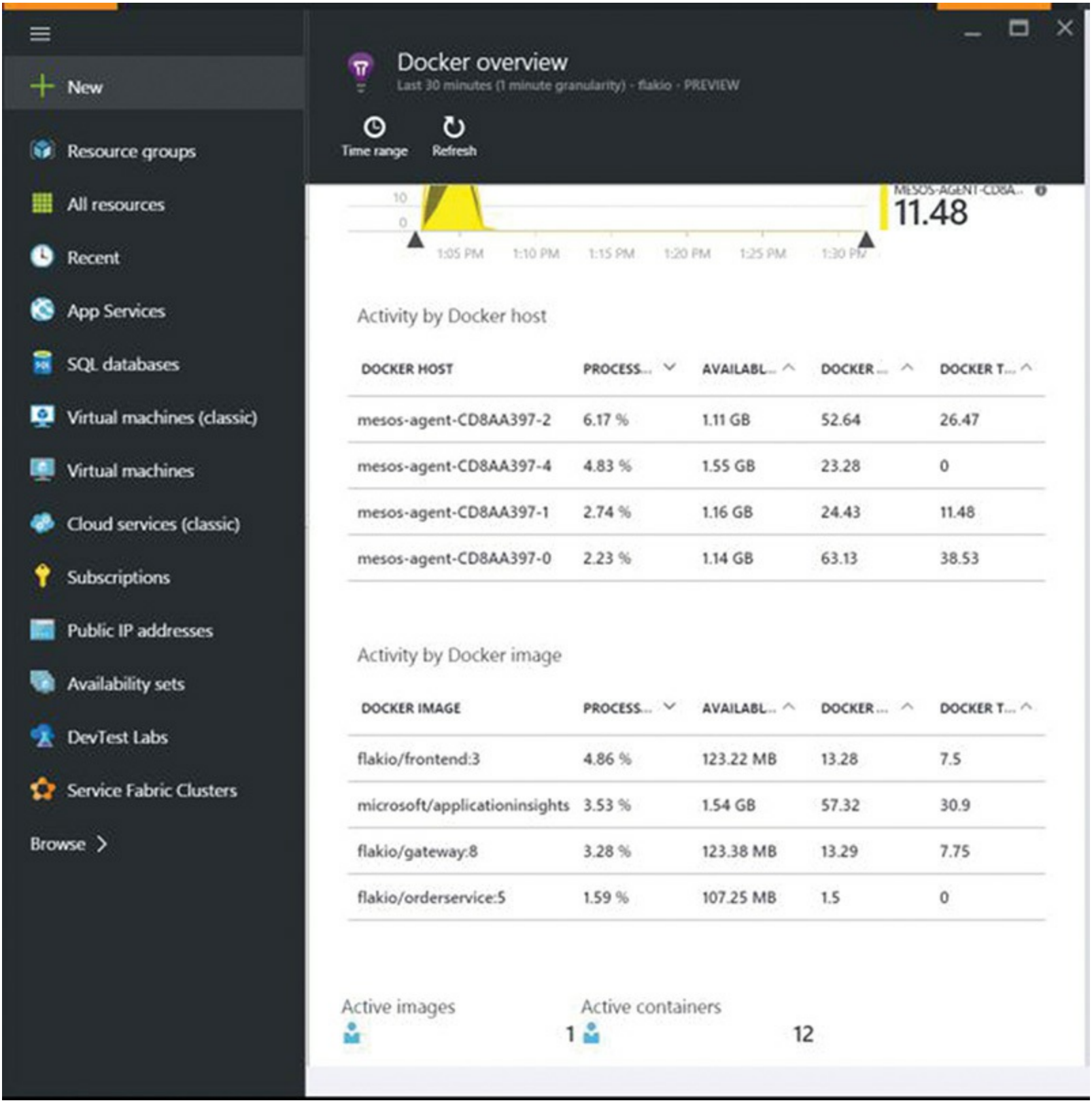

FIGURE 7.8: Application Insights Docker overview blade

The Docker overview blade is a good starting point for drilling deeper into each component (individual Docker host, images, and containers). Figure 7.9, for example, shows the Docker by host blade.

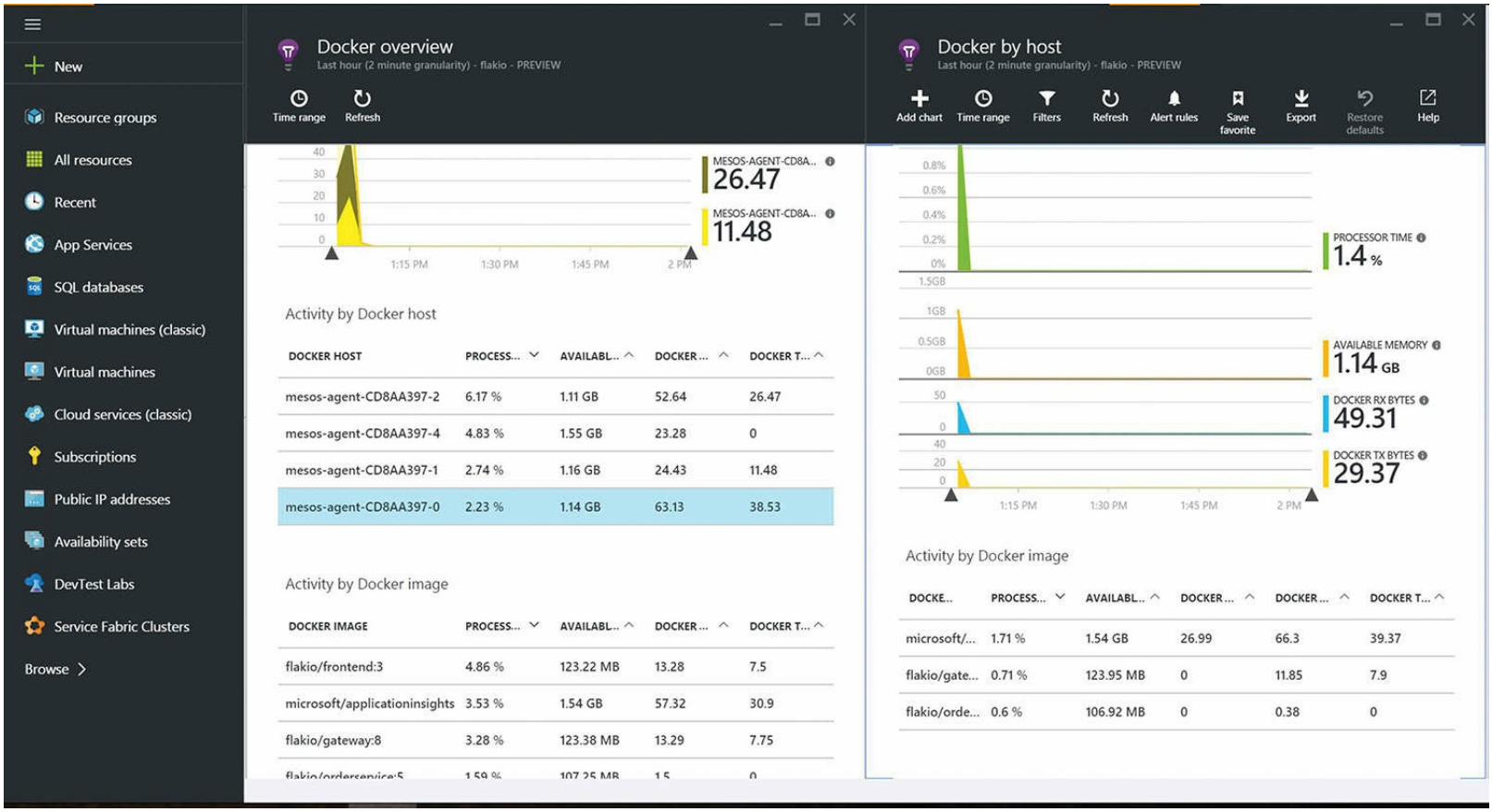

FIGURE 7.9: Docker by host blade in Application Insights

All these features make Application Insights a very powerful and developer-focused monitoring solution for containerized microservices applications on Azure.

Operations Management Suite (OMS)

The Microsoft Operations Management suite offers similar Docker-related functionality to Application Insights. It collects all the necessary log data and offers a nice UI in a standalone portal that enables users to drill into all the components of a containerized microservices architecture, such as Docker host VMs in an environment, images, and containers.

So what is the difference? Application Insights is targeting container monitoring with an option to gain additional application-level telemetry on Azure, so it is really all about the developer experience. OMS, on the other hand, is an all-in-one cloud management solution that targets public clouds such as Azure, AWS, OpenStack, and so on, as well as hybrid clouds, and is all about the management aspects of containers rather than just monitoring.

OMS offers two ways of installing and enabling its agent on host VMs.

- The agent is available as a download on MSDN and GitHub. As it is a standalone agent, we could use Puppet, Chef, or Azure Resource Manager to install and configure the agent as part of our environment-provisioning step.
- Pull the OMS agent image from Docker Hub.

As mentioned before, the Docker monitoring functionality is very similar to what Application Insights offers. We get all the monitoring data relevant to the containers, such as:

- Number of container host VMs and their utilization
- Number of running containers

- Container and image inventory
- Container performance and container logs
- Docker daemon logs

In addition to the data, OMS will also associate the application logs with relevant container information if the application logs to STDOUT/STDERR.

In addition to its Docker integration, OMS is a great solution if we have to manage resources across various clouds as it enables people to have one single portal to monitor and manage those resources.

Recommended Solutions by Docker

Below is a list of additional monitoring solutions that are part of Docker's Ecosystem Technology Partner Program.

- AppDynamics
- Datadog
- New Relic
- Scout
- SignalFx
- Sysdig

More details about the program and each solution can be found here: https://blog.docker.com/2015/06/etp-monitoring/

Summary

In this chapter, we have covered why it is important to monitor the entire microservices environment, starting from host VMs in a cluster all the way to the services running in containers. We have seen best practices and guidance for VMs, containers, and services. In addition, we looked at some monitoring and diagnostics solutions offered by Microsoft. The biggest takeaway should be that monitoring is a key part of microservices-based architectures, and that we need to include it very early in our planning.

8. Azure Service Fabric

Over the last couple of chapters, we have seen that building highly available, resilient and performant microservices requires more than just containers. We need additional system services for cluster management, services for container or service orchestration, service registration, service discovery, and service monitoring. Mesosphere DCOS or Mesos proper, Zookeeper, Marathon, Docker Swarm, and Kubernetes are currently the most popular services to help address those areas and are frequently used together. Separately within Microsoft, Azure has built Azure Service Fabric as a platform to build exactly these types of highly available, resilient, and performant microservices. It provides cluster management, service orchestration, service registration, service discovery, service monitoring, and more in a single cohesive platform, thereby eliminating the need to manage multiple systems. Many Microsoft services such as Azure DB, Cortana and Intune, to name just a few, have been using Azure Service Fabric as their platform for the last five years. Azure Service Fabric is currently available as a public preview and will be made generally available in the first half of 2016.

Service Fabric plays an important part in Microsoft's Azure strategy, and thus we need to understand what Service Fabric is and how we can use it. This chapter provides a high-level overview of Service Fabric with the goal of making us aware of its features, and how we can think about Service Fabric concepts in the context of what we have learned so far.

Service Fabric Overview

Service Fabric is a complete platform with application runtimes for easily building microservices on public Azure, private clouds, on-premise, and other configurations.

> **Linux Support**
>
> Per Microsoft's current plans, Linux support will be available in 2016. The Service Fabric functionality on Linux will be almost a hundred percent identical.

Service Fabric includes system services that take care of everything that is important for resilient and scalable microservices environments. Cluster management, resource scheduling, service registry and discovery, failover, deployment, no-downtime rolling upgrades, and safe rollbacks are just some of the platform's features. The key here is that those services are part of Service Fabric itself and do not need to be installed or configured separately—the integration has been done for us.

Service Fabric Subsystems

The Service Fabric system services are part of what Service Fabric calls subsystems. The major subsystems are:

- **Management:** The management subsystem is used to provide end-to-end service and application lifecycle capabilities, including deploying services, monitoring and diagnostics, and patching and upgrading those services without any downtime. The management subsystem can even do all the same things for Service Fabric runtime itself. Developers can use managed APIs, PowerShell commandlets, and/or HTTP APIs to access these management capabilities.
- **Health:** Service Fabric comes with its own health system. All components of the system such

as cluster, nodes, services, and so on can report their health status. The data is stored in a centralized health store like Azure storage. Services inside of Service Fabric can also add and customize the health model by emitting custom reports into the system, which Service Fabric also utilizes alongside the built-in health properties that it understands natively.

- **Communication:** The most important service of this subsystem is the naming service. The naming service provides service registry and service discovery functionality within a cluster.
- **Reliability:** This subsystem is responsible for making sure that the services in a cluster are highly available. There are three important components in this subsystem:
  - The cluster resource manager, which places service/container replicas across all nodes, ensures that placement rules are enforced, and monitors and balances resource consumption.
  - The failover manager service, which monitors nodes and services and reacts to failures and other changes in the cluster. The Failover Manager, with the help of the Replicator and Naming, also handles leader election for stateful services (we will learn more about stateful services later in the chapter).
  - The Service Fabric Replicator, which ensures consistency between primary and secondary replicas for stateful services and manages quorum. We will look more closely at replicas and quorum later in the chapter.
- **Federation:** This subsystem is responsible for managing membership of nodes in the cluster. It provides failure detection; lease management, and node join/depart semantics.
- **Transport:** This subsystem is responsible for point-to-point communication within a cluster.
- **Hosting:** Hosting is not a real subsystem from Service Fabric's perspective, but we are going to talk about it here for better understanding. Most of the system services in Service Fabric handle both cluster management and only service/container management. For our one-to-one comparison with Mesos, Kubernetes, Swarm, and so on, it is easier to think about some components as being responsible for hosting services and containers. Service Fabric also refers to this concept as Hosting.

Figure 8.1 shows at a high level how the Service Fabric subsystems map to a container-based solution on Azure.

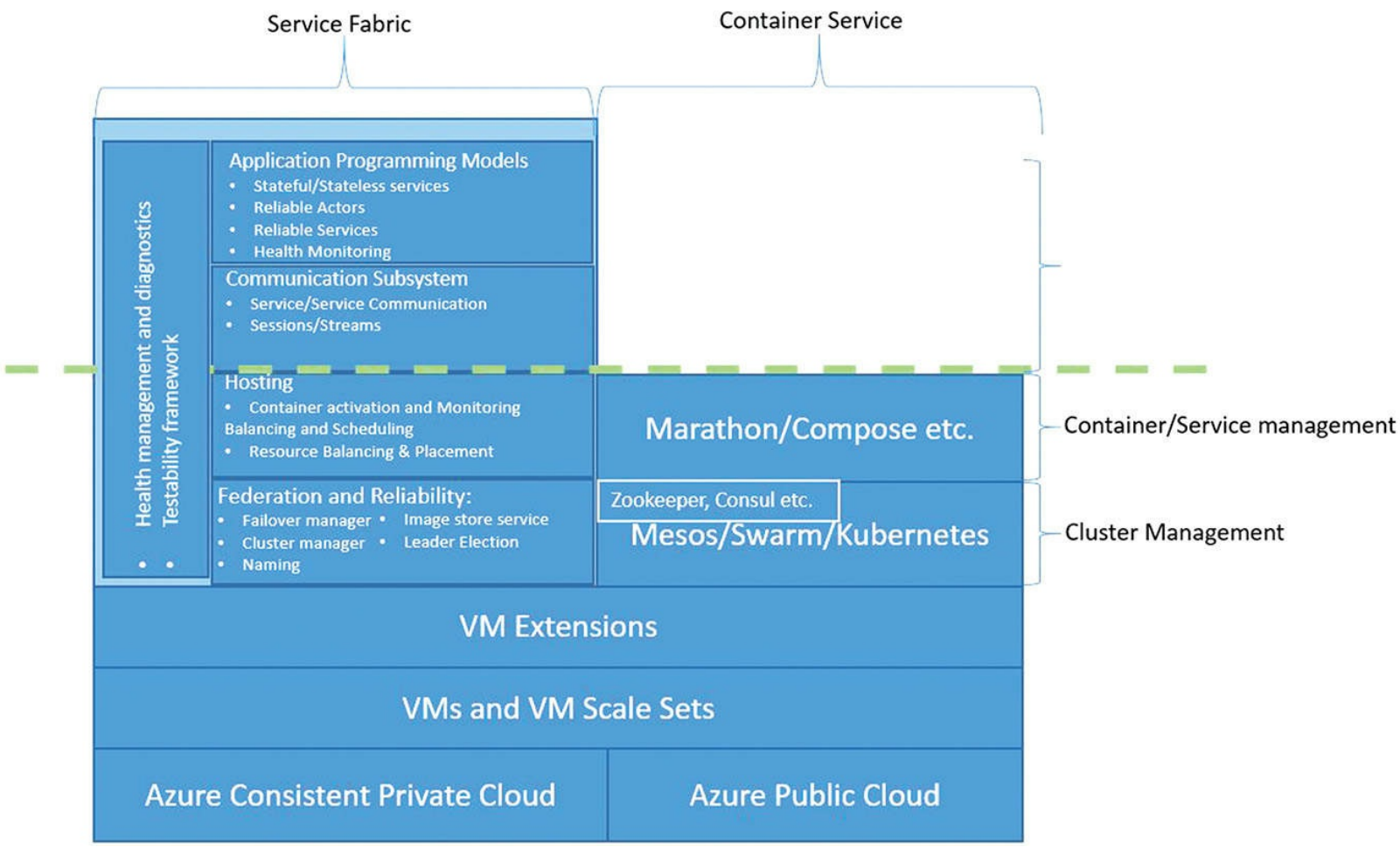

FIGURE 8.1: Service Fabric and container services

If we look at all the subsystems, it becomes clear why Microsoft positions Service Fabric as a platform for building microservices. Service Fabric not only offers full cluster and service or container management, but it also offers a full programming model for developers to leverage if they want. For the remainder of the chapter, we will have a closer look at cluster management, service and container management, and the programming model.

Cluster Management

In Service Fabric as well as in a containerized microservices architecture, there is a notion of a cluster, which consists of a number of nodes. In the context of this chapter, a node relates to a physical or virtual machine instance running Service Fabric.

In Service Fabric, a cluster is defined through the cluster manifest described in an Azure Resource Model (ARM) JSON document. The cluster manifest specifies infrastructure metadata such as number of nodes, node names and types, network topology, cluster security, and so on. One big difference compared to a Docker Swarm or Mesos cluster is that Service Fabric does not require us to set up a separate service for managing cluster availability with tools like Zookeeper or Consul. These are normally needed for leader election among several machines, which then manage the cluster. Service Fabric is a single stack designed specifically to work together vs. a coordinated set of open-source projects that are used together. As a result, we only need to specify the nodes needed to run our microservices in the ARM template, and the Service Fabric cluster bootstraps itself from there. To ensure the highest-availability clustering, Service Fabric uses the quorum model and requires us to use at least five nodes for a minimal cluster when setting up a cluster in Azure.

What is Quorum?

Quorum is a term used in high-availability clustering and means that there needs to be a minimum number of nodes online to be able to communicate with each other. Of those, in most cases a majority need to agree on a piece of information for it to be considered "the truth." In the case of Service Fabric Quorum has to be established between the replicas of a stateful service.

At the time of writing, Azure Service Fabric was still in preview mode and was using Azure virtual machines for its nodes.

Virtual Machine Scale Sets

For general availability, Service Fabric will use virtual machine scale sets to take advantage of its integrated dynamic scale features. This blog post at https://azure.microsoft.com/en-us/blog/azure-vm-scale-sets-public-preview/ has great information for virtual machine scale sets.

To make sure that not all virtual machines can go down at the same time—for example, for planned maintenance—Service Fabric Service uses Azure availability sets. Every virtual machine in an availability set is put into an upgrade domain and is spread across fault domains by the underlying Azure platform.

Fault and Update Domains

Fault domain: A fault domain is a physical unit of failure consisting of hardware components such as host machines, switches, and so on, that share a single point of failure. A server rack is a good example of a fault domain. By default, Azure spreads an availability set over two fault domains. Service Fabric, however, is using a minimum of three fault domains to offer the highest possible degree of availability.

Upgrade domain: An update domain is a logical unit of a set of nodes that can be updated together. In Azure, an availability set contains twenty update domains.

Figure 8.2 shows a simplified model of how Service Fabric sets up a five-node cluster across upgrade and fault domains.

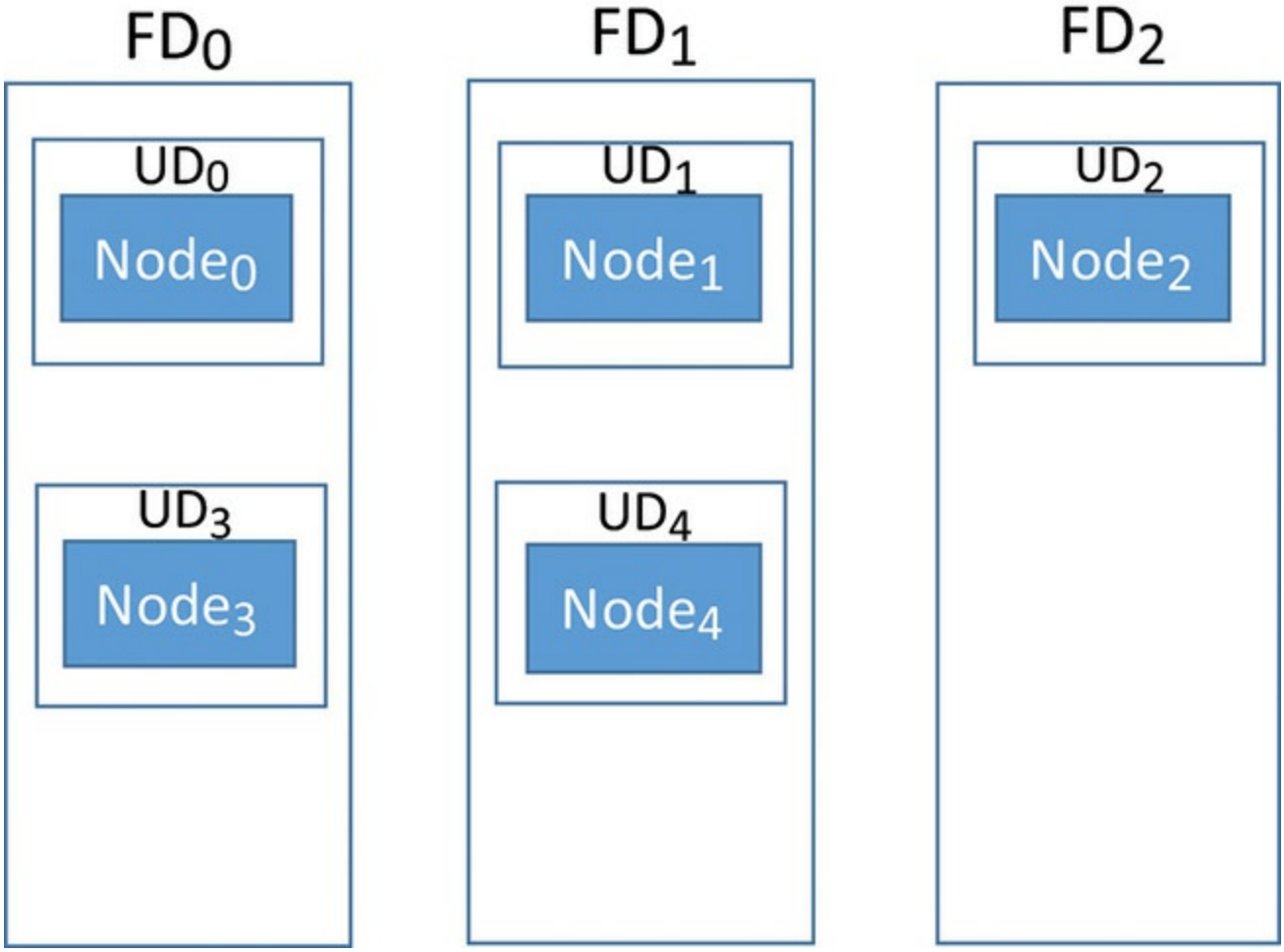

FIGURE 8.2: Five-node cluster across upgrade and fault domains

For clusters with more than five nodes, Azure continues to put the nodes into unique fault domains, upgrade domain combinations using more than just three fault domains. Azure will follow this pattern for any number of nodes. Figure 8.3 shows an example distribution of VMs across fault and upgrade domains in a seven-node cluster.

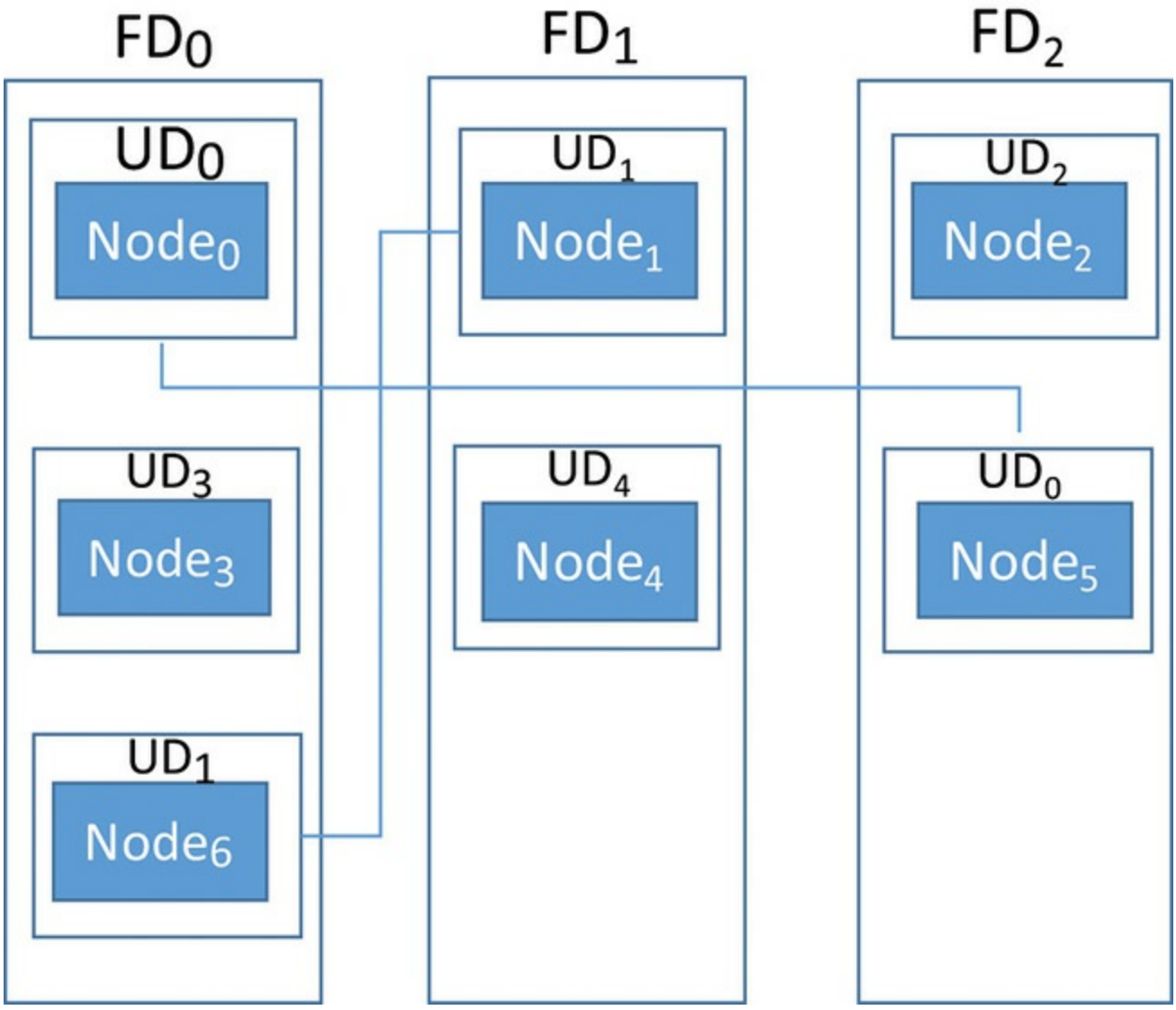

FIGURE 8.3: Seven-node cluster across upgrade and fault domains

Now we know what a Service Fabric cluster is and how it works at a high level; we just need to know how to set one up. Service Fabric clusters in Azure can be set up like any other Azure Resource Manager-based environment. We can use the Azure portal, CLI, or PowerShell to provision a Service Fabric cluster. Figure 8.4 shows how to create a Service Fabric cluster through the Azure portal.

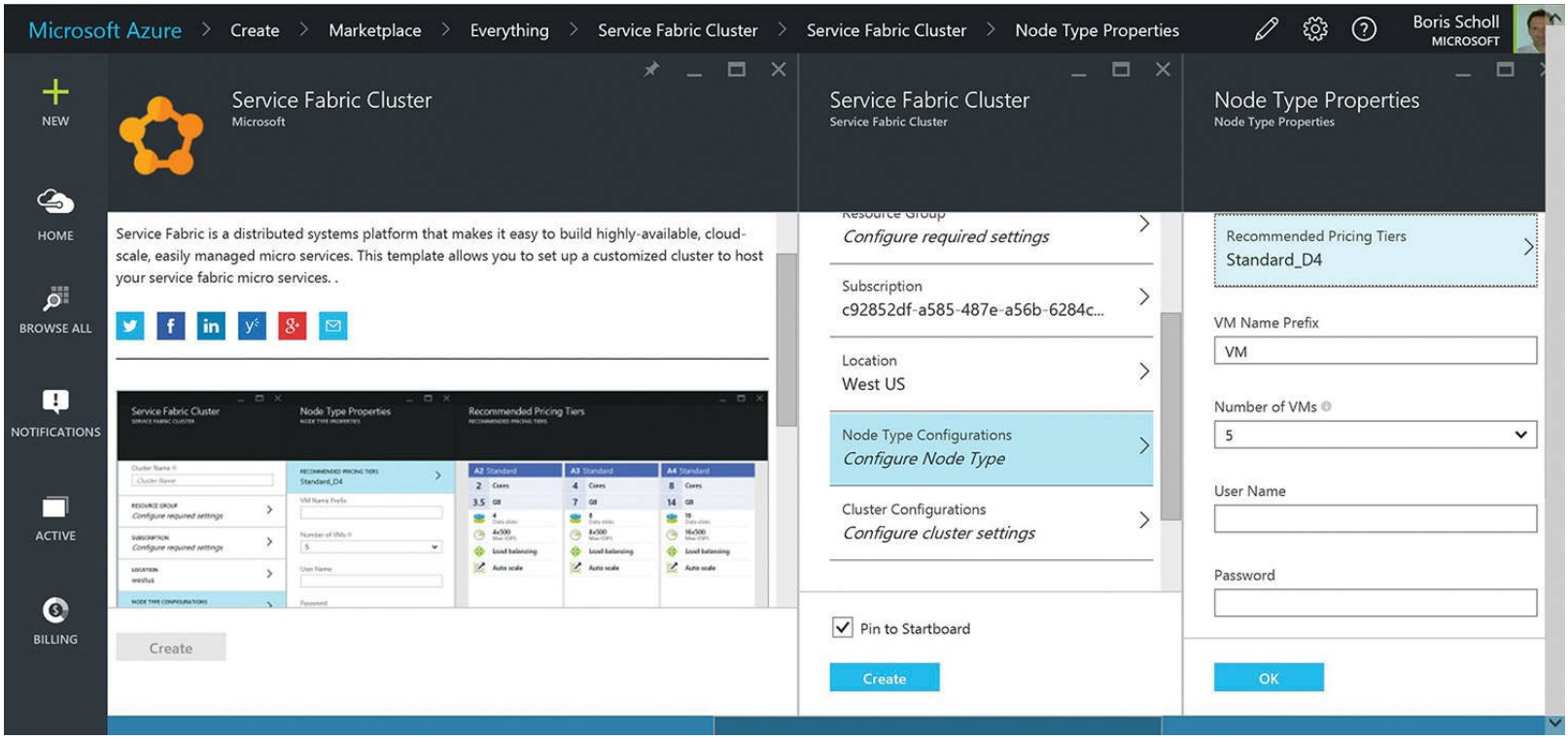

FIGURE 8.4: Create a cluster through the Azure portal

Resource Scheduling

As we know from our experience with Mesos, we can use schedulers such as Marathon, Kubernetes, or Docker Swarm to orchestrate containers that contain services and place them on the nodes in the cluster. Each scheduler has its pros and cons: for example, at the time of writing Swarm does not automatically place containers on a different node if you scale down your cluster. As most schedulers or orchestrators are open-source, we can expect those functionalities to improve over time. Placing containers or services on machines is also often referred to as resource scheduling.

Service Fabric Application

To understand how Service Fabric handles resource scheduling, it is important to understand the concept of an application in Service Fabric. A Service Fabric application is a collection of constituent services. Each service consists of code, configuration, and data packages that implement the functionality of that particular service, which is illustrated in Figure 8.5.

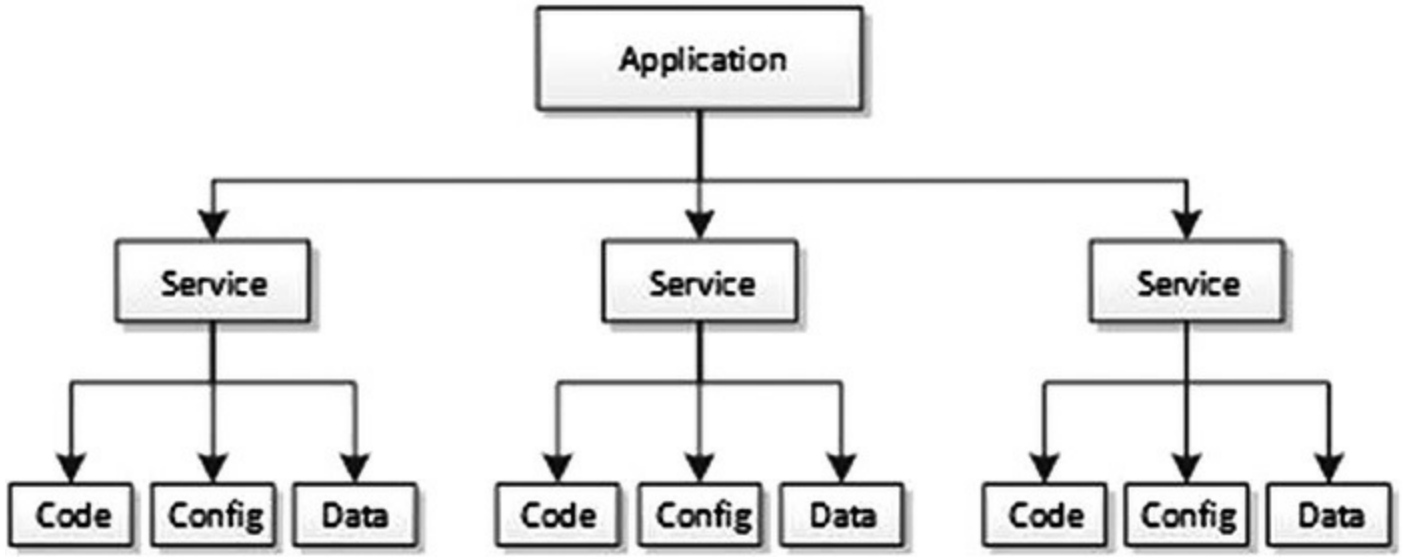

FIGURE 8.5: Service Fabric application structure

Application Manifest

An application is defined through its manifest file. At the time of writing, the application manifest is a definition file authored by the application developer, which describes the physical layout, or package, of a Service Fabric application. The application manifest references the service manifests of the constituent services from which the application is composed. From a conceptual point, the application manifest can be compared to the docker-compose.yml file, which is used to define and run multicontainer applications with Docker. In addition to referencing the services in an application, developers can use the application manifest to configure the "run-as" policy as well as the "security access" (that is, read/write-on resources) for each imported service when the application is instantiated. This is typically very useful to control and restrict access of services to different resources to minimize any potential security risk.

Service Manifest

The services referenced in the application manifest are defined through a service manifest authored by the service developer. The service manifest specifies independently upgradable code, configuration, and data packages that together implement a specified set of services that can then be run in the cluster once it is deployed.

The service manifest is used to define what service or containers need to be executed, what data belongs to a service, what endpoints to expose, and so on, which is very similar to a Dockerfile where developers define OS, frameworks, code, and endpoints to build a Docker image which is used for a container. In the service manifest, developers can configure additional things like service types (stateful or stateless), load metrics, and placement constraints.

> **Note**
>
> With placement constraints, developers can make sure that services or containers are only placed on a certain node type. Node types can be configured in the ARM templates and contain the name of the node type and its hardware configurations. A good example is that developers can make sure that their web services only run on nodes that are tagged with the node type "webservers." The nodes of type "webserver" could be based on virtual machines with a hardware configuration optimized for web server workloads.

See https://azure.microsoft.com/en-us/documentation/articles/service-fabric-application-model/ for more information on application and service configuration.

Once the application is running, Service Fabric starts automatically resource-balancing the services across the underlying shared pool of cluster nodes to achieve the optimal load distribution, based on a set of load metrics using the services available in the Hosting subsystem. In case of a node failure, Service Fabric moves the services that were running on that node onto the remaining nodes. The placement happens based on available resources and service metrics. Another great functionality of Service Fabric is that it rebalances the services across the cluster in the case of scale-out in a cluster. We will learn more details about that later in the chapter when we look at stateful services.

Custom Applications (Existing Applications)

A custom application is an application that does not integrate with Service Fabric, meaning it does not reference any Service Fabric binaries. Examples of custom applications are Node.js applications, Java applications, MongoDB, and so on. Benefits of running custom applications in Service Fabric are high availability of the application, health reporting, rolling upgrade support, and higher density.

- **High availability:** Applications that run in Service Fabric are highly available out of the box. Service Fabric makes sure that one instance of an application is always up and running.
- **Health monitoring:** Out-of-the-box Service Fabric health monitoring detects if the application is up and running and provides diagnostics information in the case of a failure.
- **Application lifecycle management:** Besides no downtime upgrades, Service Fabric also enables rolling back to the previous version if there is an issue during upgrade.
- **Density:** You can run multiple applications in a cluster, which eliminates the need for each application to run on its own hardware.

Custom applications are packaged and deployed in the same way as Service Fabric applications. The service manifest contains additional metadata that tells Service Fabric where it can find the binaries of the application and how to execute it. Once the package is copied into the image store, Service Fabric pulls down the application package to the nodes and executes the custom applications. This article provides more information for deploying custom applications: https://azure.microsoft.com/en-us/documentation/articles/service-fabric-deploy-existing-app/. One can easily see that instead of starting custom applications, Service Fabric could spin up Docker or Windows containers, rather than just inside processes and Windows Job Objects as they do currently.

Container Integration

Container Support will be available in preview around the first half of 2016, Service Fabric will support Windows, Hyper-V, and Docker containers. There will be two levels of integration:

- **Bring your own container:** In this scenario, Service Fabric will handle containers the same way as custom applications. The service manifest defines all the settings, such as image, ports, volumes, and others that are needed to launch a container. In this scenario, Service Fabric becomes a first-class orchestrator for containers.
- Develop and package Service Fabric services inside containers. In this scenario, developers will be able to create new stateful and stateless services and package them inside a container.

Service Discovery

As we have seen in previous chapters, there needs to be some sort of service registry and discovery service that enables services to make themselves known to the system but also enables them to know where to find other services they need to talk to and to expose them to the outside world. Tools and services like Eureka, Zookeeper, or Consul, coupled with things like HAProxy are popular ones in the Docker microservices world. Service Fabric has service registration and discovery included through its Naming service. As mentioned before, the Naming service provides name resolution of services running somewhere in the cluster. Service instances and replicas get an opportunity to register whatever address they want with the Naming Service, and Name resolution takes the stable service name and transparently maps it to that address, so that the service is always addressable no

matter on which node it runs in the cluster.

Programming Model

Service Fabric does not stop at the cluster and container/service management level; it also comes with its own programing models. Before we look at the programming models, we should understand that Service Fabric supports both stateless and stateful services.

Stateless Services

The definition of stateless is that something is truly stateless where the data is only stored in memory; a calculator is a good example of a stateless application. Stateless can also mean that the data is persisted in an external storage solution, such as SQL DB or Azure Tables. This is the standard model for distributed systems today, with tiered architectures being the most popular. To be able to enable our service to scale out by adding more instances, we avoid storing any state on our service itself. To make stateless services resilient, we need to implement lots of patterns when accessing state in the backend, starting with proper queuing patterns to pass messages between the tiers, through caching patterns to improve performance, and ending with sophisticated retry patterns to make sure we can retrieve the data from external systems.

Stateless services in Service Fabric are pretty much the same as any other types of stateless services such as Azure Cloud Services or web apps. We can make those services highly available and scalable by starting more instance of the service. A typical scenario for a stateless service is a gateway service, which accepts the incoming traffic and routes the requests to the stateful services. Another example is a simple web front end that provides a UI for any type of user interaction.

Stateful Services

One important pattern we need to implement for high-performing services is the colocation pattern. Colocation means that the data needs to be as close as possible to the service to avoid latency when accessing the data. In addition, the data is usually partitioned to avoid too much traffic on a single data store, which then becomes a bottleneck.

This is exactly where we can use stateful services in Service Fabric. Think of stateful services as those where the data is partitioned and colocated directly within the service itself. Service Fabric ensures reliability of data through replication and local persistence.

There are three important concepts when it comes to stateful services replication.

- **Partition:** A partition is a logical construct and can be seen as a scale unit that is highly reliable through replicas, which are spread across the cluster.
- **Replica:** A replica is an instance of the code of the service that has a copy of the data. Read and Write operations are performed at one replica (called the Primary). Changes to data due to write operations are replicated to multiple other replicas (called the Active Secondary replicas). This combination of Primary and Active Secondary replicas is the replica set of the service. Service Fabric places each replica in a set on a different node across fault and upgrade domains in the cluster to improve scalability and availability. It also redistributes them in the case of a cluster scale-out or scale-in to ensure an optimal distribution.
- **Replication:** Replication is the process of applying state changes to the primary and secondary replicas. A replica is an object that encapsulates the state of a failover unit. In other words, it

contains a copy of the services code and the state. All replicas that back up a stateful service build a replica set. Service Fabric places each replica in a set on a different node across fault and upgrade domains in the cluster to improve scalability and availability.

Figure 8.6 shows a cluster with two partitions and their replicas spread across the nodes.

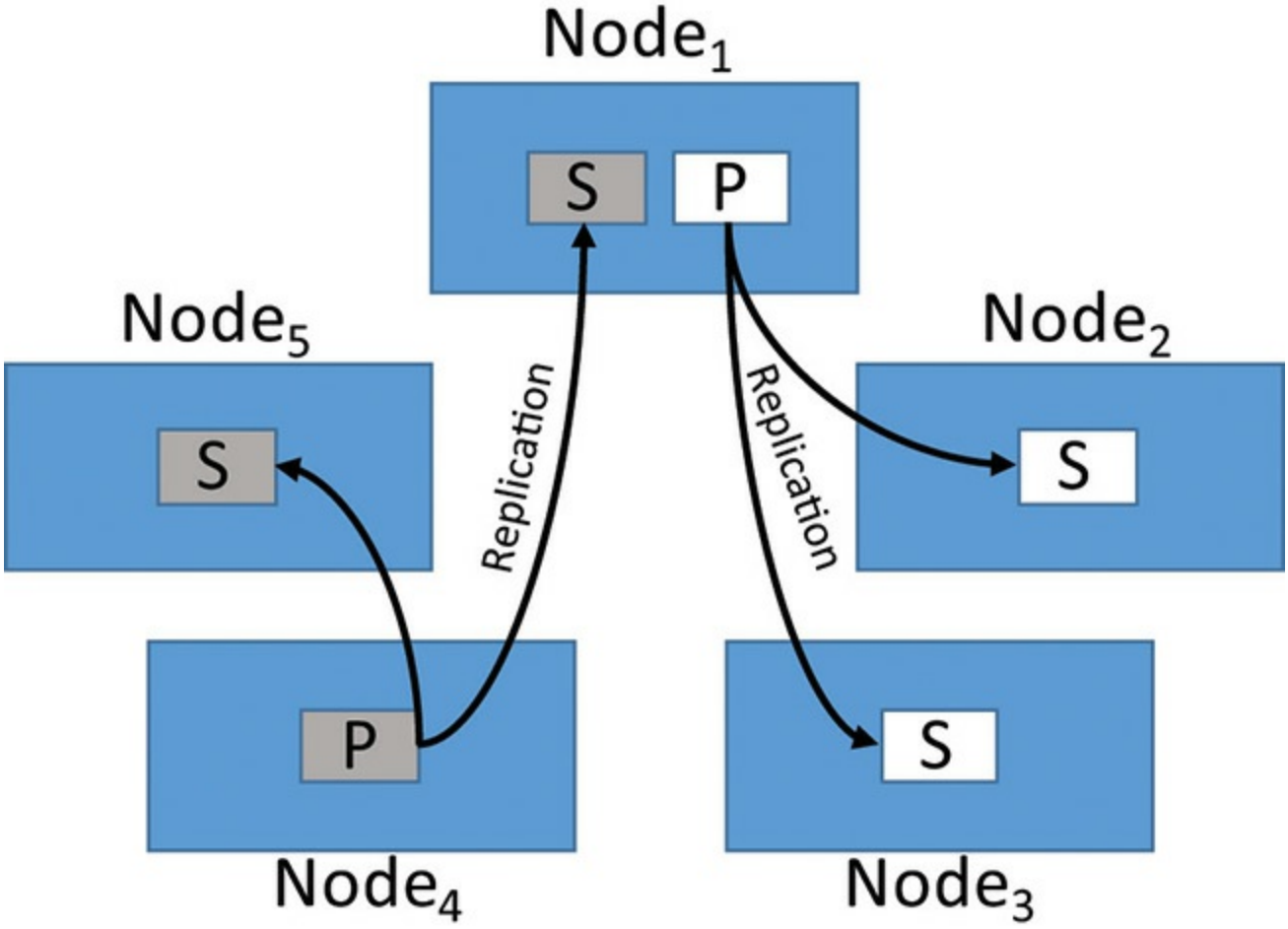

FIGURE 8.6: Five-node cluster with two partitions

So how does this work practically? Let's assume we have implemented a stateful service that calculates the digits of PI. In this case, Service Fabric would create a primary replica on some node (say Node 1) and secondary replicas on other nodes like Node 2 and Node 3. The primary replica receives all reads and writes, in our case the calculated digits and the state of the execution, and synchronizes the state with the secondary replicas. In case Node 1 goes down, it would automatically failover to one of the secondaries. As each secondary contains the state, the service could continue with the calculation once the new primary, for example Node 2, is up and running. As soon as Node 1 is up again it will be made a secondary. That sounds like a lot of work and we might wonder what we need to do as developers? The good news is that we need to do *nothing* as Service Fabric handles all this for us.

This article provides detailed information on how to partition stateful services: https://azure.microsoft.com/en-us/documentation/articles/service-fabric-concepts-partitioning/.

Besides resiliency and performance, we gain some more advantages when building stateful services. First, we can eliminate dependencies on external systems and with that reduce potential points of failure. Second, we do not need to implement all the code for accessing external systems, including retry handlers, and similar items. Third, we can reduce the number of components such as Cache we had to use in a traditional three-tier architecture. Fourth, it makes it a lot easier to deploy a complete system in a development environment because there are fewer external dependencies to

manage.

Now that we know what kind of services and applications we can build in Service Fabric, we can have a look at the programming models. Service Fabric has two main programming models.

Reliable Actors

Reliable Actors API: Actors were introduced in the 1970s to deal with concurrent computation problems in parallel and distributed computing systems to simplify programming. From a logical point of view, actors are isolated, single-threaded, and independent units of compute and state.

> **Concurrency Problems**
>
> In computer sciences, the dining philosophers' problem is a great example to illustrate concurrency and synchronization issues that can be solved using actors. Wikipedia has a good article on this subject at https://en.wikipedia.org/wiki/Dining_philosophers_problem, which is worth reading if someone is not familiar with those types of problems.

With the advent of cloud computing, there are more and more use cases where actor-based applications offer a great advantage. Almost every application with multiple independent units of state, such as online games or IoT (Internet of Things) scenarios, are great use cases for actors. In the case of online games most of the times an actor represents a player, or in the case of IoT, an actor represents a device.

Service Fabric implements reliable actors as virtual actors, which means that their lifetime is not tied to their objects in memory. The advantage of virtual actors is that we do not need to explicitly create or destroy them. The Service Fabric actor runtime automatically activates the actors when needed and garbage collects them if they have not been used for a while. Once garbage collected, the actors are not really "gone," as the runtime still maintains knowledge about their existence. The runtime also distributes the actors across the nodes in the cluster, and with that achieves high availability in the same way as we have seen earlier in this chapter. So how does that help with concurrency? Service Fabric only enables one active thread inside an instance of the actor code at any time. The runtime requires a complete execution of an actor method, for example as a response to requests from other actors or clients even though the communication is asynchronous. The complete execution is called a *turn*, and thus we speak of turn-based concurrency. Service Fabric makes it easy to use actors as the runtime does all the heavy lifting. As developers we just need to define the actor interfaces, implement them, register the actor implementation, and connect using an actor proxy, and the runtime will do all the heavy lifting, like synchronization and making the state of the actor reliable for us.

Reliable Services

Reliable Services API: We have learned that developing reliable services in Service Fabric is different from developing traditional cloud services, as Service Fabric services are highly available and scalable "out of the box" without writing any additional code. How to develop stateless services based on the reliable services APIs is pretty straightforward and is outside the scope of this chapter.

However, when developing stateful services, Service Fabric offers a programming model that is

unique: the Reliable Collections API. Reliable collections enable us to develop highly available microservices. The local and replicated state is managed by the reliable collections itself. This is a big deal as developers can simply use reliable collections as collection objects as if they are only targeting single machines.

The biggest difference to other high-availability technologies, such as Redis cache or Service Bus queues, is that the state is kept locally in the service instance, and with that eliminates the latency of reads across a network. As we have learned before, the state is replicated and with that, our service is highly available when using reliable collections. More information on reliable collections can be found here: https://azure.microsoft.com/en-us/documentation/articles/service-fabric-reliable-services-reliable-collections/.

To conclude the Programming Model section, it is worth mentioning that Service Fabric ships an SDK that offers full Visual Studio integration, a local development experience and great diagnostics capabilities for .NET development. At the time of writing, the development experience was restricted to .NET development, but the team is working on supporting Java and other languages. We can expect a preview in the first half of 2016. Figure 8.7 shows the create new Service Fabric application experience in Visual Studio.

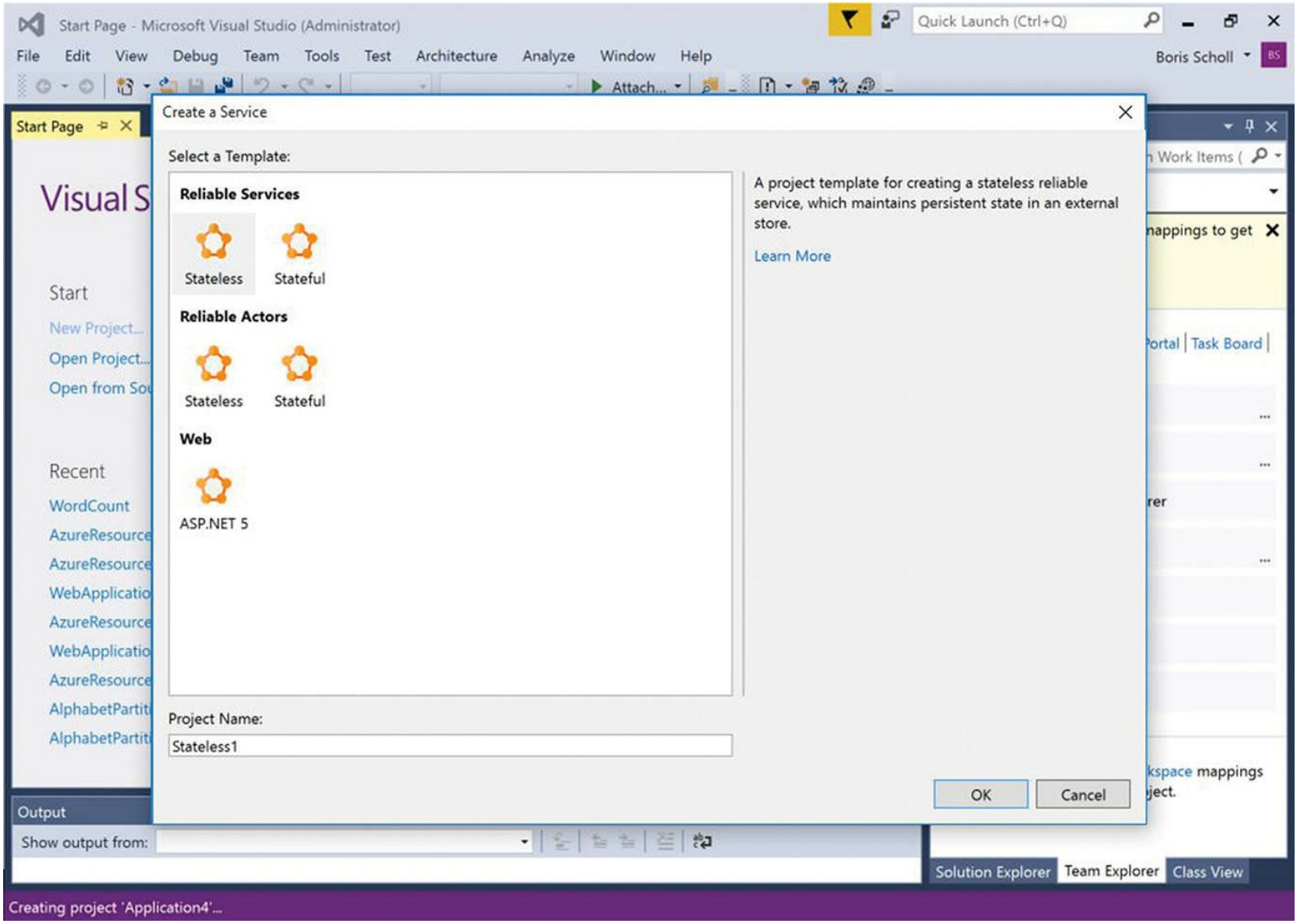

FIGURE 8.7: Create new Service Fabric application in Visual Studio 2015

In addition to the development experience, Service Fabric provides a modern Service Fabric Explorer that gives us all the insights into the cluster, application, and service status. The Service

Fabric explorer is part of SDK and you can launch it from a Windows tray icon. Figure 8.8 shows the Service Fabric explorer on a development machine.

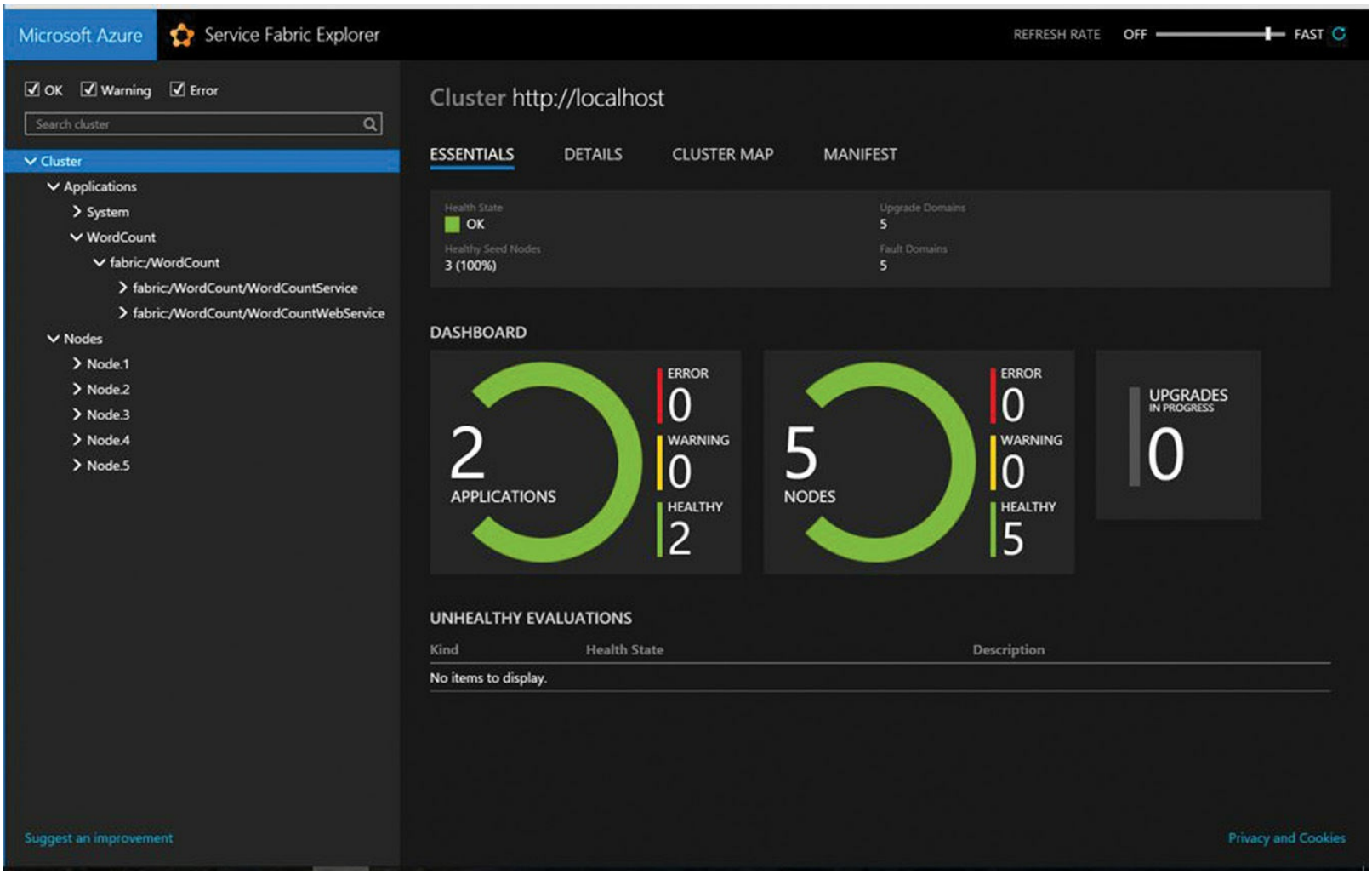

FIGURE 8.8: Service Fabric explorer on a development machine

Service Fabric Explorer is also available on any cluster in Azure. Every time we provision a new cluster in Azure, its setup can be configured on the nodes in the cluster. It listens to port 19080.

Application Lifecycle

Throughout this book, we have learned that one of the key advantages of microservices is that one can update and upgrade microservices with no downtime of the running version. Service Fabric comes with great application lifecycle capabilities that enable us to update services, upgrade services, and test services.

Service Updates

Properties of running services can be updated through PowerShell, REST Endpoints, or the object model. A good example for updating a service is to add more instances to a running stateless service or change the replica counts of running stateful services.

Application Upgrades

Service Fabric enables us to upgrade each service in an application independently, which is right in line with the notion of independently versionable and deployable microservices. Let's assume we have an application with a stateless service that serves as the web front end, and a stateful service

that serves as our back-end services. As we have learned, both services are referenced in the application manifest and each service has its own service manifest. Assume further that we have made updates to the backend service. In this case, we would only need to change the version number of the application itself and the version of the back-end service. The front-end service can remain unchanged and in its initial version.

The upgrade itself is done as a rolling upgrade. This means that the service is upgraded in stages. In each stage, the upgrade is applied to the nodes in a specific upgrade domain. By continuing upgrades in that manner, Service Fabric ensures that the service remains available throughout the process. Figure 8.9 shows the upgrade process of an application in Service Fabric Explorer, with the service in upgrade domain 0 completed and in upgrade domain 1 in progress.

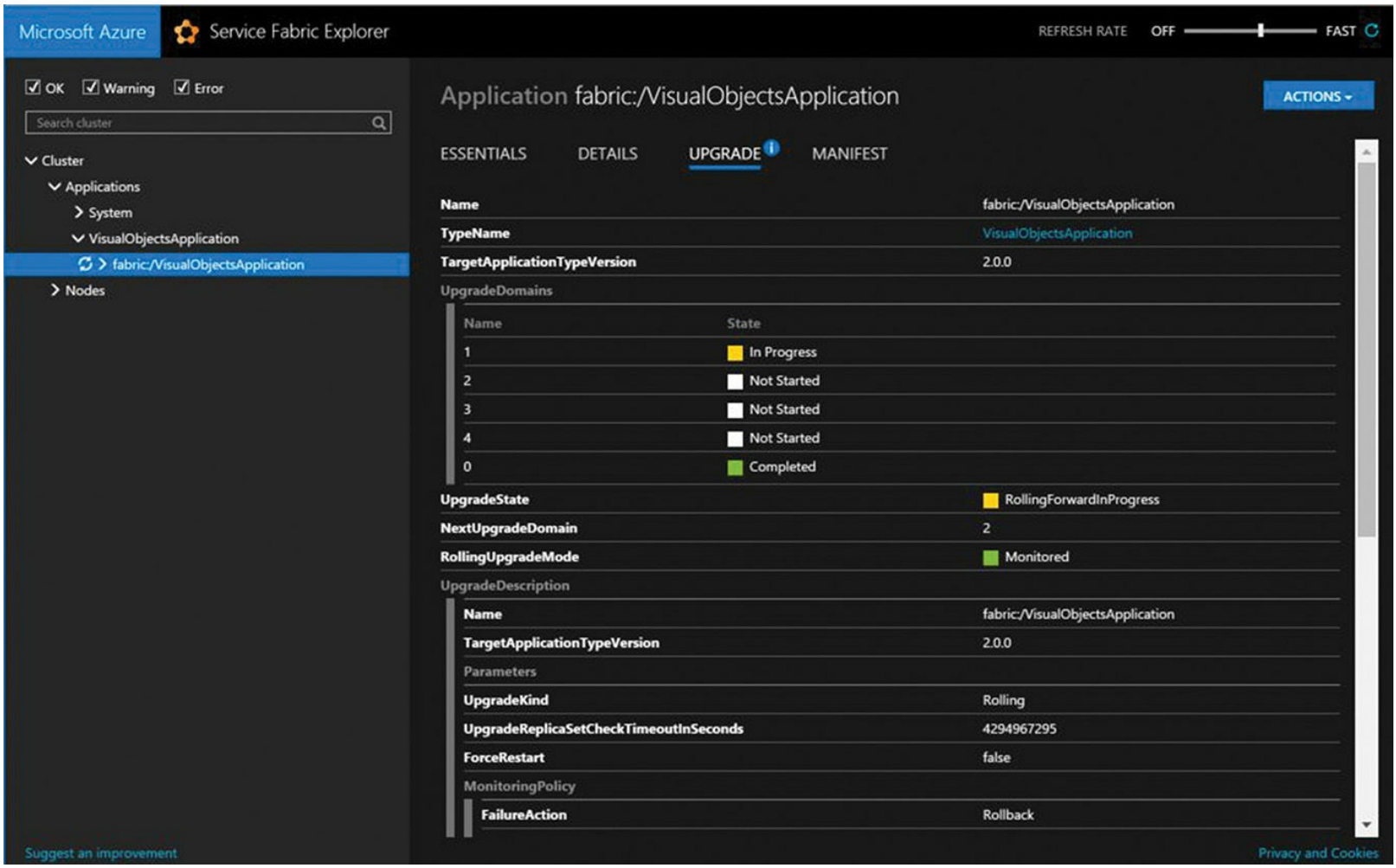

FIGURE 8.9: Upgrade status of an application in Service Fabric Explorer

During the entire upgrade, the process is monitored by default. The upgrade is performed on one upgrade domain, and if all health checks pass, moves on to the next upgrade domain automatically, and so on. If the health checks fail and/or timeouts are reached, the upgrade is rolled back for the upgrade domain.

More information on Service Fabric upgrades and health checks can be found here: https://azure.microsoft.com/en-us/documentation/articles/service-fabric-application-upgrade/.

Testability Framework

As we have seen in Chapter 6, testing plays an important role in the DevOps cycle of microservices development. Whereas code-centric tests like unit tests are still up to the developer, Service Fabric supports testing applications and services for faults in a number of ways.

The simplest and easiest way to test our service in case of node failures is using the Service Fabric Explorer. It provides an easy way to shut down, stop and restart nodes. Those types of tests enable us to see how a service behaves under those circumstances and thus provide valuables insights. When testing stateful services, we can usually observe how Service Fabric promotes a secondary to become the primary while the service is still available and keeps its state. Figure 8.10 shows the action menu that enables us to shut down and restart the node.

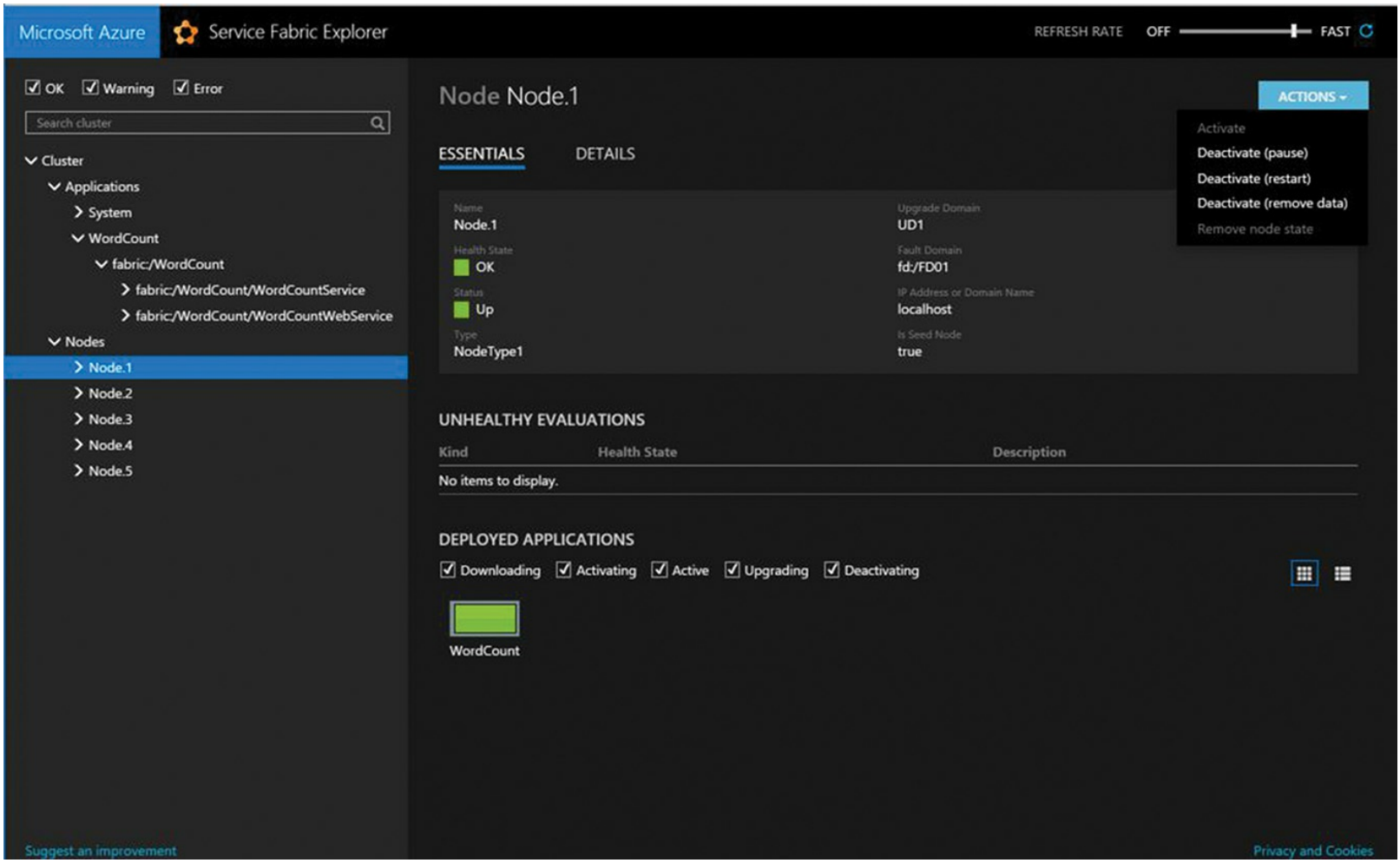

FIGURE 8.10: Service Fabric Explorer node action menu

Besides this simple test, Service Fabric also comes with a more advanced testability framework that enables us to conduct Chaos tests and Failover tests. We could write an entire chapter covering the Service Fabric testability framework but that is beyond the scope of this book. For us it is important to know that such a testability framework exists and that tests can be executed in an automated manner through C# and PowerShell.

More information about Service Fabric testability framework can be found here: https://azure.microsoft.com/en-us/documentation/articles/service-fabric-testability-overview/.

Summary

As we have seen, Service Fabric is a complete framework for building microservices-based applications. In addition to its great cluster management and resource scheduling capabilities, it offers a rich programming models that make it easy for developers to create highly available, resilient, and high-performing services. Perhaps the biggest and most compelling aspect of service fabric is that it democratizes the development of stateful and highly available services. At the time of writing, the Service Fabric Service was still in preview and some changes will certainly happen, but

the core concepts of Service Fabric described in this chapter will remain the same.

A. ASP.NET Core 1.0 and Microservices

This appendix provides an overview of the key design changes coming with the newest version of ASP.NET including a new execution environment, cross-platform support, configurable environment variables, and more. We'll then review some common best practices and considerations for building and designing containerized microservices architectures.

A New Version of ASP.NET

ASP.NET Core 1.0, previously known as ASP.NET 5, represents the biggest fundamental shift in how ASP.NET applications work since the original release of ASP.NET 1.0 in 2002. For developers and architects evaluating a microservices architecture, it's important to understand the new capabilities of ASP.NET and how to take advantage of them in a microservices architecture. This chapter isn't designed to give you an overview of everything that has changed in ASP.NET Core 1.0, which could be its own book, but rather to give you an overview of fundamental features you need to know about.

Getting Started

You can find installation and setup instructions for Mac, Linux, and Windows as well as additional documentation at http://docs.asp.net.

Open Source Stack

One of the biggest changes to ASP.NET and the .NET Framework, the runtime used by ASP.NET is that ASP.NET Core 1.0 and .NET Core are open-source projects, meaning the pieces of the stack, from collections classes to MVC to the underlying runtime execution environment, are all open source. The benefits are obvious in that the community is able to contribute to the stack, either in the form of new features or bug fixes, or even using the source code to help debug framework classes.

Cross-platform ASP.NET Apps

For developers evaluating a containerized architecture, the biggest fundamental change for ASP.NET is that ASP.NET Web and Console applications can run on either a Linux or Windows Docker host. Previously, ASP.NET applications required the full .NET Framework to be installed, and of course could only work on Windows. Every ASP.NET Core 1.0 project now explicitly declares what frameworks it supports, and you can support both the full .NET Framework and the .NET Core in a single project.

.NET Core

.NET Core is a lightweight, cross-platform, modular version of the full .NET Framework that includes CoreFX, a set of .NET libraries (collections, threading, data access, common data types, and so on), and the CoreCLR, the underlying runtime and execution environment that handles assembly loading, garbage collection, type safety, and other features. One of the key benefits of .NET Core's modularity is that all your app's dependencies, including the CoreCLR, can be packaged and distributed as standalone items. That means you no longer have to install the full .NET Framework on a server to run ASP.NET apps. You can find more information on .NET Core at http://bit.ly/aspdotnetcore.

.NET Core Side-by-Side Support

.NET Core's architecture also alleviates one of the most common difficulties reported by customers, where their server environment is locked into a specific version of the .NET Framework that cannot be upgraded because of the potential to break legacy apps. With .NET Core, all the dependencies of the app are included in the app itself. This enables you to have three different apps that reference three different versions of the Core CLR without causing side-by-side compatibility issues. Your legacy apps can continue to use an older version of the Core CLR without restricting your newer apps to an outdated version of the Core CLR.

Application Dependencies using project.json

Similar to the NodeJS model, ASP.NET provides a new configuration file named project.json that lives in the root of your project folder. Project.json describes a project's dependencies, including target frameworks as discussed in the section on the cross-platform ASP.NET apps, NuGet libraries that your project depends on, how and what to include and exclude when building for output, command line tools like scaffolding to run, and pre-compilation tasks for Javascript and CSS for your project. The key takeaway is that it's as easy as updating a JSON file to target a new framework or add a new NuGet library, and that you can do this directly from any text editor, without the need to use Visual Studio.

Everything is a NuGet Package

.NET developers have been using NuGet, the .NET package manager for reusable libraries, for years to easily update, distribute, and discover libraries. .NET Core takes this further by requiring that all.NET components be packaged, distributed, and updated through NuGet. Previously, developers and administrators had to wait about 12 to 18 months for new functionality in the .NET Framework to be available. Now, because .NET Core libraries ship as NuGet packages, you can instantly add or update the libraries or the underlying runtime your application uses. For example, the code snippet here, from the dependencies section of the project.json file shows a list of dependencies. Each line item represents a NuGet package based on the namespace and the corresponding version of that library, which at the time of this writing is Release Candidate 1 (rc1).

Click here to view code image

```
  "dependencies": {
    "Microsoft.AspNet.Mvc": "6.0.0-rc1-final",
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final",
    "Microsoft.Extensions.Configuration.FileProviderExtensions":
"1.0.0-rc1-final",
    "Microsoft.Extensions.Configuration.Json": "1.0.0-rc1-final",
    "Microsoft.Extensions.Logging": "1.0.0-rc1-final",
    "Microsoft.Extensions.Logging.Console": "1.0.0-rc1-final"
...
```

It isn't just the .NET Framework that is packaged using NuGet; any class library that you build for .NET Core is also packaged and distributed as a NuGet library. In fact, if you use Visual Studio's **Add Reference**... feature to reference a class library in your solution, in ASP.NET Core 1.0, what it does is add a new dependency to the list of dependencies in your project.json file.

Cloud-ready Environment-based Configuration

Application configuration, stored as key/value pairs like service URLs, connection strings, application constants, or other configuration settings, are now based on a flexible cascading configuration system, with providers for JSON, ini files, and environment variables. The reason this is important for containerized solutions is that if you remember from earlier chapters, when we create a Docker image we are copying or "hard-coding" the contents of our web project into that image. But what happens if there are some things in your project, like a database connection string, that you really don't want to hard-code in? Docker solves this by providing a way to dynamically inject environment variables into a container when it's created. ASP.NET's support for environment variables makes this perfect in that we can hardcode a connection string in the image and dynamically replace the value when we create the container in a development, staging, or production environment.

To set up a project's configuration, as in previous versions of ASP.NET, you can use the startup.cs file to add code to setup the configuration services you want to use in the **Startup** constructor. The example below shows adding two configuration services. The first one reads configuration settings from a file named config.json. The second reads configuration settings using environment variables. The order of configuration providers matters as the order dictates the precedence rules to overwrite the same configuration found in multiple providers. In the list below, this means that environment variables will take precedence and overwrite any matching configuration in the config.json file.

Click here to view code image

```
public Startup()
{
  Configuration = new Configuration()
  .AddJsonFile("config.json")
  .AddEnvironmentVariables();
}          ...
```

In the sample config.json file below, notice that you can go beyond just the key/value pair and build a set of nested configuration settings like the ConnectionStrings section that includes a child database connection string.

Click here to view code image

```
{
  "AppSettings": {
    "SiteTitle": "My Website"
  },
  "ConnectionStrings": {
    "SqlDbConnection":
"Server=(localdb)\\mssqllocaldb;Database=Products;Trusted_
Connection=True;MultipleActiveResultSets=true"
  }

}
```

To load the connection string, you can then just use the **Configuration** object that is created in the `Startup` constructor and pass in a string that follows the nested hierarchy for the configuration setting as shown below.

Click here to view code image

```
var conn =
Configuration.Get<string>(["ConnectionStrings:SqlDbConnection");
```

You definitely don't want to add a production database string to source control for the production environment, so you instead set the connection string using an environment variable that is set when a Docker container is created. You can see an example of using ASP.NET and Environment variables in a Docker container in Chapter 4, "Setting Up Your Development Environment," or refer to Chapter 6, "DevOps and Continuous Delivery" to read about more advanced options for handling environment configuration changes using services like Consul or Zookeeper.

Command-line Driven

ASP.NET is also command-line driven, meaning that common tasks, from scaffolding to build and packaging, to deployment, can be done from either the command line or Visual Studio. Doing this also enables you to use your favorite code editor, including cross-platform code editors for ASP.NET development, instead of requiring Visual Studio for development. At the time of this writing, the .NET and ASP.NET teams are rebranding their command line tools from "DNX" (DotNet eXecution) to "dotnet." The .NET Core Command Line Interface (CLI) includes the following commands for managing .NET projects:

- **Compile:** Compiles your project.
- **New:** Creates a new "Hello World" .NET console project template.
- **Publish:** Prepares your project for deployment by downloading any dependencies, including the .NET Core Framework that the project requires to run.
- **Restore:** Restores NuGet dependencies for your project.
- **Run:** Compiles and runs your project.

Unification of ASP.NET MVC and Web API

In previous versions of ASP.NET, ASP.NET MVC 5 and Web API 2 had similar but different implementations for things like controllers, actions, filters, model binding, routing, and more. In ASP.NET Core 1.0, MVC and Web API have been unified into MVC, which provides a single way to handle routing, controllers, model binding, and so forth.

Dependency Injection as a First Class Citizen

Dependency Injection, or DI for short, is used to easily swap out project-wide components in a standard way. Let's say you have a data access library. Rather than create a concrete class for that data access library, you can create an interface for data access and swap out providers. For instance, you could do this to use a mock library for local development versus a real database configured to use SQL Server, or to change to a wholly different data provider, such as using MongoDB by simply implementing the interface contract and changing the provider in the startup.cs file. As discussed in Chapter 6, dependency injection becomes especially important for testing dependencies between different microservices.

Cross-platform Data & Nonrelational Data with Entity Framework

As data access is one of the most popular uses for ASP.NET applications running on Linux or Mac, being able to access non-Windows databases like MySQL, PostgreSQL, or SQLite is critical. The Entity Framework team is working with the community to provide drivers that enable cross-platform data access. Going further, EF is also working on support for nonrelational data stores like NoSQL

stores, including MongoDB, Azure Table Storage, and more. You can find the current status of EF drivers at http://bit.ly/aspefdb.

RAD Development using Roslyn

For developers, one of the most-loved improvements is that ASP.NET has a dynamic compilation system where you no longer need to build/compile your source code. Instead, ASP.NET Core 1.0 will dynamically compile your source code using "Roslyn," the .NET Compiler platform. What this means for a developer is that they can hit F5 to run their app, switch to source code to make a change, save the code changes, and refresh the browser to see those changes instantly. That's right, no need to stop the debugger, click **Build**, and then F5 again!

Cross-platform Web Server

Kestrel (https://github.com/aspnet/KestrelHttpServer) is the name of ASP.NET's open-source cross-platform (Windows, Mac, and Linux) web server, based on libuv (libuv.org), an asynchronous IO library. Kestrel is packaged as a NuGet library and can easily be run on the command line to listen and respond to web requests from a specified port (default 5004). IIS, the built-in Windows web server, is still available for Windows hosting, but for Mac or Linux you must use Kestrel for handling web requests.

Cross-platform Console Applications

One of the most hidden features of ASP.NET Core 1.0 is the added support for console applications which are designed to run on Linux, Mac, and Windows. Console applications make it easy to build system services or "headless" processes that interact with the underlying Linux, Mac, or Windows operating system. .NET developers now have an incredibly easy way to build their own or to reuse any existing Linux-based command-line-driven tools like FFmpeg for video manipulation, network tools like Nmap, and more by calling them from a cross-platform console application.

What's Not Included?

It's important to note what's not possible with ASP.NET Core 1.0. In particular, ASP.NET Web Forms is not part of ASP.NET Core 1.0, and having it will still require the full .NET Framework installation. While ASP.NET 4 apps will not run on Linux, you can still run them Dockerized within a Windows Server 2015 Container with the full .NET Framework installed.

Choosing the Right ASP.NET Docker Image

There are a few things to consider when deciding what underlying execution environment you want to use for your ASP.NET application. For Linux, your choices are .NET Core and Mono. At the time of this writing, Mono is a more mature and fuller-featured framework, and more third-party NuGet packages support Mono than .NET Core. That said .NET Core is quickly catching up in terms of feature parity, and with Mono you lose .NET Core's benefits, like side-by-side support. This becomes less of an issue for containers as each Dockerfile could include a different Mono runtime and you can have multiple containers, each with different versions of the Mono framework installed on the same Docker host.

The common recommendation for Linux is that .NET Core is the better choice in the longer term and that Mono, in general, isn't the best option for production environments because of the lack of full

.NET framework parity, and the energy and investments currently being made in .NET Core.

For Windows, there are a couple of considerations. The first is whether to run on Windows Nano Server or full Windows Server. While most developers are familiar with Windows Server, Nano Server is a new headless, remotely managed version of Windows Server that has been dramatically trimmed down, resulting in better resource utilization, tighter security, and a smaller overall footprint. It includes a number of benefits like a much smaller size, 80 percent fewer reboots, and improved security. If you are running on Windows Nano Server, you must target .NET Core because the full .NET Framework isn't supported.

For developers migrating existing applications to ASP.NET Core 1.0, the first thing you'll want to check is whether the NuGet packages you depend on have a compatible version that runs on .NET Core. Every NuGet package you depend on needs to explicitly target .NET Core. For new ASP.NET applications, .NET Core is a simpler, lightweight solution, and in general is the recommended approach for the future. Table A.1 summarizes the different runtime and operating system choices for ASP.NET.

	Operating System		
Runtime	Linux	Windows Nano Server	Windows Server 2016
Mono	Yes	No	No
.NET Core CLR	Yes	Yes	Yes
.NET Full Framework	No	No	Yes*

* Includes ASP.NET 4.x apps running in a container.

TABLE A.1: Runtime and Operating System Options for ASP.NET

Visual Studio 2015 Tooling

Support for Modern Front-end Web Development Practices

ASP.NET and Visual Studio 2015 also now support modern front-end web development practices, meaning they include integration with JavaScript package manager (think NuGet for JavaScript) like Node Package Manager (NPM) and Bower, the package manager for client frameworks like Bootstrap and Angular. The new tooling also includes support for automated build and compilations steps, like Grunt or Gulp. Grunt or Gulp are popular task runners that, similar to a build system for server code, do the same build optimization steps for client-side JavaScript, like minifying your JavaScript by removing whitespaces and carriage returns. For developers building Docker images, this is important as you'll need to ensure that any pre-compilation tasks happen as part of the Docker image creation process as discussed in Chapter 4. These tasks can be specified in a project.json scripts section that installs NPM and Bower, and then runs Gulp tasks to clean and minify content.

Click here to view code image

```
  "scripts": {
    "prepublish": [ "npm install", "bower install", "gulp clean",
"gulp min" ]
  }
```

The default minify task, for example, calls two tasks:

Click here to view code image

```
gulp.task("min", ["min:js", "min:css"]);
```

The min:js task uses the built-in uglify() method to convert readable javascript into obfuscated code. The CSS task, on the other hand, minifies the CSS contents to reduce the amount of bytes sent over the wire. You don't have to understand how these tasks work, but the important part is to ensure that these tasks run as part of the Docker image creation, either as part of the normal Visual Studio publishing process, or via an automated build and continuous integration process.

Click here to view code image

```
gulp.task("min:js", function () {
    gulp.src([paths.js, "!" + paths.minJs], { base: "." })
        .pipe(concat(paths.concatJsDest))
        .pipe(uglify())
        .pipe(gulp.dest("."));
});

gulp.task("min:css", function () {
    gulp.src([paths.css, "!" + paths.minCss])
        .pipe(concat(paths.concatCssDest))
        .pipe(cssmin())
        .pipe(gulp.dest("."));
});
```

Easily Build REST Proxies for Microservices

One of the coolest and least-known features available in Visual Studio 2015 is the capability to create a client proxy based on Swagger metadata for a REST service. Swagger is a standard, language-agnostic way to describe your REST APIs. A Swagger file includes information about your API's custom data types, inputs, return types, resources, and more. This has the benefit of making your APIs self-describing so developers and tools can understand how to use your APIs. It also means that tools can automatically create client proxies in Java, .NET, Node, Python, Ruby, and a host of other languages for a Swagger service. For .NET developers, Visual Studio includes the capability to provide a URI to the Swagger metadata, and it will build a custom proxy available as source code in your project for you to call the service. If this sounds familiar, it is similar to SOAP (Simple Object Access Protocol) and WSDL (Web Service Description Language), which provided similar capabilities more than a decade ago. The difference with Swagger is that REST services are inherently lighter-weight and simpler than a SOAP service, and for SOAP services the generated proxy from tools was provided as a DLL, meaning that a developer could not add to or modify the code. With auto-generated Swagger classes, the source code for the service is provided to you so you can add, edit, or modify it to your heart's content.

ASP.NET Microservices Best Practices

As you can see, there are a lot of new improvements and enhancements to ASP.NET for developers and architects to consider. In addition to framework- and feature-level improvements, we also

wanted to add some generally accepted recommendations or considerations when designing microservices using ASP.NET Web API.

Use Async Everywhere

Ensure that your web APIs use the Async/Await pattern where possible. When executing a request synchronously, for example an I/O request to read a file, the thread will idle and block execution until the I/O request is complete. Using the Async/Await pattern enables your thread to return to the thread pool so it can handle future requests. For API developers, ensure that both your controllers and anything that accesses resources, files, database repositories, other APIs, and so on, are designed to use Async and Await. Here's a basic before and after example that assumes a data repository named **DbContext**. In the second example, the Async controller method is marked with the `async` keyword, and the call to the `ToListAsync()` method is prefixed with the `await` keyword.

Click here to view code image

```
[HttpGet]
public IEnumerable<Product> Get()
{
    return DbContext.Products;
}
[HttpGet]
public async Task<IEnumerable<Product>> Get()
{
    return await DbContext.Products.ToListAsync<Product>();
}
```

Design Stateless APIs

While this design principle has been obvious for the last several years for web developers, it's especially important with the introduction of containers. Remember that containers are ephemeral, meaning that if you write an API to upload photos that in turn saves photos to the local disk inside the container, those images by default will be gone when the container is stopped or removed (unless you set up a Docker volume mount). Stateless APIs also make it easy to scale up or down the number of containers handling requests without having to worry about the state of each API. There are, however, some scenarios where you want to build stateful microservices. In those cases, Service Fabric, which supports ASP.NET, is the right choice to manage distributed state for services.

Consider the Right Web API Return Type

When designing your microservices using the ASP.NET Web API you have two general categories of return types, POCO (Plain Old C# Object) return types, and results that implement the IActionResult interface.

POCO object return types refer to types that return a C# object as shown in the next example, which returns an IEnumerable<Product> containing the first 25 products. The object is serialized into JSON and sent over wire.

Click here to view code image

```
[HttpGet]
public async Task<IEnumerable<Product>> Get()
{
    var result = await _context.Products
        .Take(25).ToArrayAsync();
```

```
        return result;
    }
```

An IActionResult return type like ObjectResult behaves exactly like the C# return type, but with the difference that with an IActionResult, you can return RESTful values using built-in primitives like HttpNotFound(), which translates to a HTTP 404 error, or return any HTTP status code using the HttpStatusCodeResult class. The snippet below shows the same API returning 25 objects, but this time using an IActionResult return type.

Click here to view code image

```
[HttpGet]
public async Task<IActionResult> Get()
{
    var result = await _context.Products
        .Take(25).ToArrayAsync();

    return new ObjectResult(result);
}
```

One common question is when to use which return type. There are no hard-and-fast rules, and although both are commonly used, the IActionResult is often used due to the additional flexibility and support for HTTP status codes it provides.

Another consideration is whether to allow for XML as a valid return type or not. Many online services like Facebook and Twitter have switched to only provide JSON for results because of the bad performance and overhead with using XML. Because XML is more verbose, it also results in higher egress costs from Azure and is potentially more expensive for clients consuming services over metered connections, such as a mobile phone. If you don't have legacy apps that depend on XML already, it's likely best to switch to only supporting `JsonResult` types.

Design RESTful APIs

When designing your APIs, use ASP.NET's attribute-based routing. As you can see in the example below, you can specify attributes for **HttpGet**, **HttpPost**, **HttpPut**, **HttpPatch**, **HttpDelete** and more. Similarly for return types, notice how we are using IActionResult and using Http return codes like the HTTP 404 request, which is returned in the **HttpNotFound()** example below.

Click here to view code image

```
public class ProductsController : Controller
{
    [HttpGet("{id}")]
    public async Task<IActionResult> Get(int id)
    {
 var product =
  await _context.Products.FirstOrDefaultAsync(p =>
     p.ProductId == id);

        if (product == null)
        {
            return HttpNotFound();
        }

        return new JsonResult(product);
    }
```

Here's a list of some of most common HTTP error codes to consider for your APIs.

1. **200 – OK**
2. **201 – Created:** A new resource was created.
3. **301 – Moved Permanently:** This resource has been moved to a new URI.
4. **400 – Bad Request:** There was an error in the request, typically caused by a malformed request (missing or invalid parameters, etc).
5. **401 – Unauthorized:** Authentication is required for accessing the resource.
6. **403 – Forbidden:** The server is refusing to respond to the request, even after authenticating it.
7. **404 – Not Found:** The given resource was not found on the server.
8. **500 – Internal Service Error:** A generic error message when a server returns an error.
9. **503 – Service Unavailable:** The service is currently unavailable say because of high load. Some Azure services, like storage, will return a 503 Service Unavailable error intermittently if they cannot handle the load.

For reference, Wikipedia includes the full list of HTTP status codes at: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes.

Consider Using New High-Concurrency Data Types

For microservices and cloud developers, one of the previous challenges with the .NET Framework has been that the default collection classes didn't include support for concurrency. This is important if, for instance, you are building a real-time "voting" microservice that enables customers to vote on something, and you'll be updating and displaying vote counts quickly. Rather than building your own custom locking code for the .NET default collections, you can use the new **System.Collections.Concurrent** namespace, which includes common data types you could use including a `ConcurrentDictionary` (key/value pairs), `ConcurrentQueue` (first-in, first-out collection), and `ConcurrentStack` (last-in, first-out collection) for improved performance.

Design your Microservice Defensively

Even if your microservice is only used by other teams inside your organization, assume that any publicly exposed API will be "fair game" to be used and abused. Defensive design of microservices means a number of things. First, you shouldn't expose API calls that are, by design, a drain on your resources. As one simple example, we can refactor the get request of our product's API so that instead of returning hundreds of products on an API call, to return a maximum of 25 results using `the LINQ` **Take()** extension method as shown below.

Click here to view code image

```
        [HttpGet]
        public async Task<IEnumerable<Product>> Get()
        {
            return await
_context.Products.Take(25).ToListAsync<Product>();
        }
```

You can then ensure that your APIs are factored to provide support for sorting, paging, and item count for further control by your callers.

Beyond API design, you also want to keep a close eye on how your API is used, who are the callers, and what API caller pattern are they using? Ensure that you have monitoring and diagnostic tools to track and understand who is using your API and how are they are using it. Depending on how your API is used (or abused), one potential option is to rate-limit your API to restrict certain callers to a fixed amount of API calls per day.

Consider a Batch API Service

Unlike your standard Web APIs, a Batch API service is not designed to return results in real time. The most common use case for a Batch API is reporting. Let's say that an internal audit team needs to make a request for a large data set, perhaps the order history for the last six months. Rather than having the audit team make hundreds of calls to your real-time APIs, you can create a separate set of Batch APIs where jobs are placed into a queue and run during system downtime, commonly late at night. The advantages of this are that you can separate out API calls that don't have to happen in real time from those that do, you can provide custom APIs for batch data requests, and you can scale each independently. One other common feature for batch APIs is support for additional return types. Adding support for comma-separated values (CSV) as a return type makes importing and exporting data into reporting tools or other automated batch processing tools much easier.

Design for Mobile Clients

In 2014, as shown in Figure A.2 from the Kliener Perkins Caufield and Biers Internet Trends report (http://www.kpcb.com/internet-trends) for the first time, more adults used mobile devices than desktop or laptops or other connected devices for digital media. Many development shops are already facing the need to build multimode applications that could run in a desktop locally via a browser, or in a custom mobile application for iOS, Android, or Windows. For web developers, this means ensuring your layout is responsive for multiple form factors, from mobile devices to tablets to desktops.

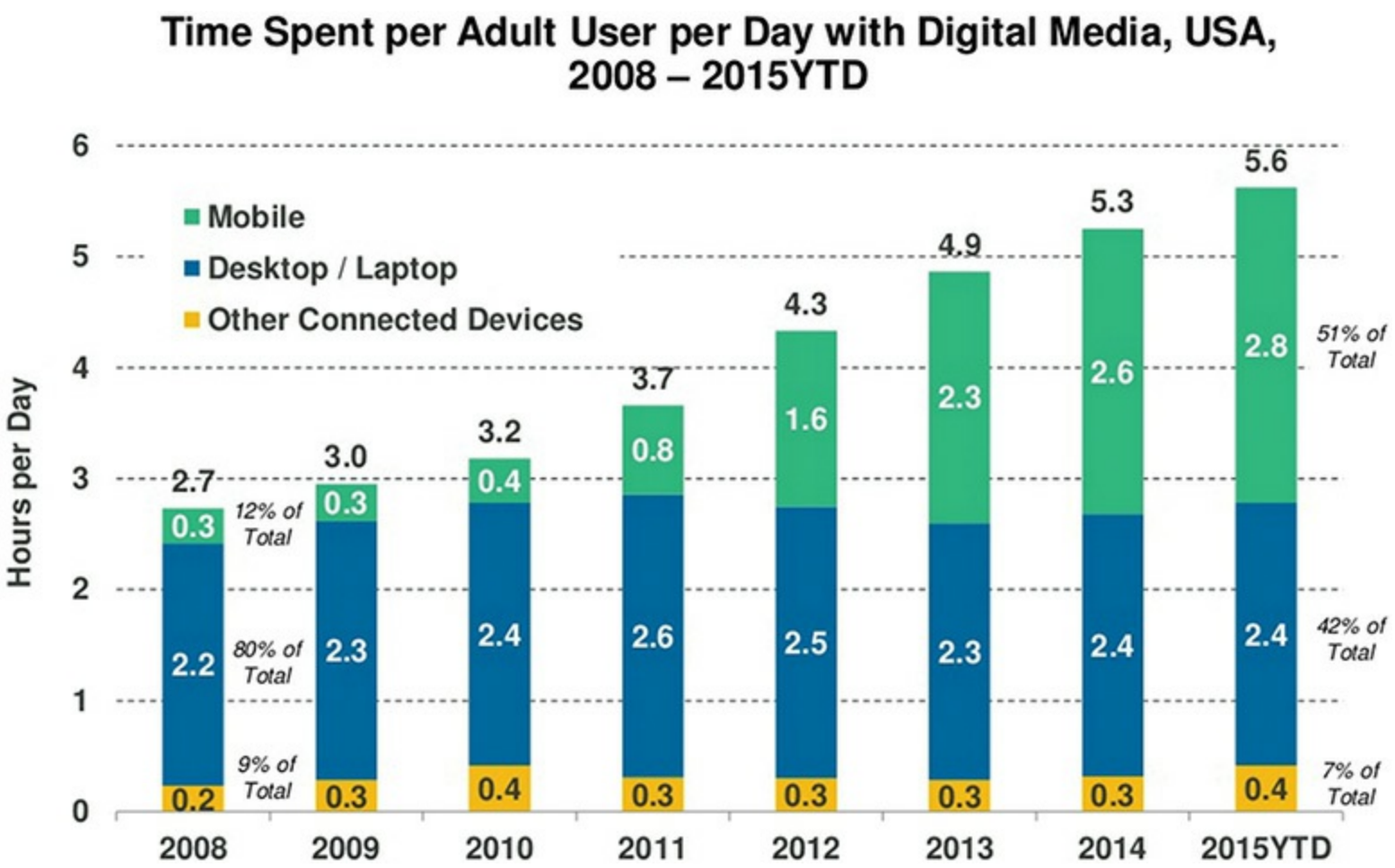

FIGURE A.1: Mobile is now used more than desktop/laptop or other devices

Consider OpenID for API authentication

Certain API requests, such as a customer's order history, should only be available after authenticating and authorizing the request. There are many ways to handle authentication, and details depend on what underlying user authentication service you are using. For example, if you are building a microservices app for your enterprise, you might want to consider using Azure's Active Directory service to handle user authentication. For Active Directory authentication, you can use OpenID Connect, a standard and extensible identity layer on top of the OAuth 2.0 protocol standard. With OpenID, when an unauthorized request to an API is received, the request is forwarded to Active Directory for a user to sign in, for instance from an iOS device. After successfully authenticating, the user receives a token that it can then use for calling APIs that require authentication. Choosing and designing the right authentication system is a book unto itself, but to get started, the Active Directory team publishes a number of examples on their team GitHub for integrating OpenID Connect with ASP.NET Web and Web API projects, Node, iOS, and many more at http://bit.ly/azureadsamples.

Make Your Microservices Easier to Consume Using Swagger

If your goal is to increase the number of developers using your microservice inside your enterprise, the onus is on you to make it as simple as possible to consume. Swagger, as we've discussed earlier, is a way to describe your APIs so that a consumer of your service can open a browser, see what services you have available, and even make simple test requests for your service. Companies such as Facebook, which provides a rich set of APIs for your social graph, provide a similar tool to test-drive their API with the Graph API Explorer as shown in Figure A.2, https://developers.facebook.com/tools/explorer.

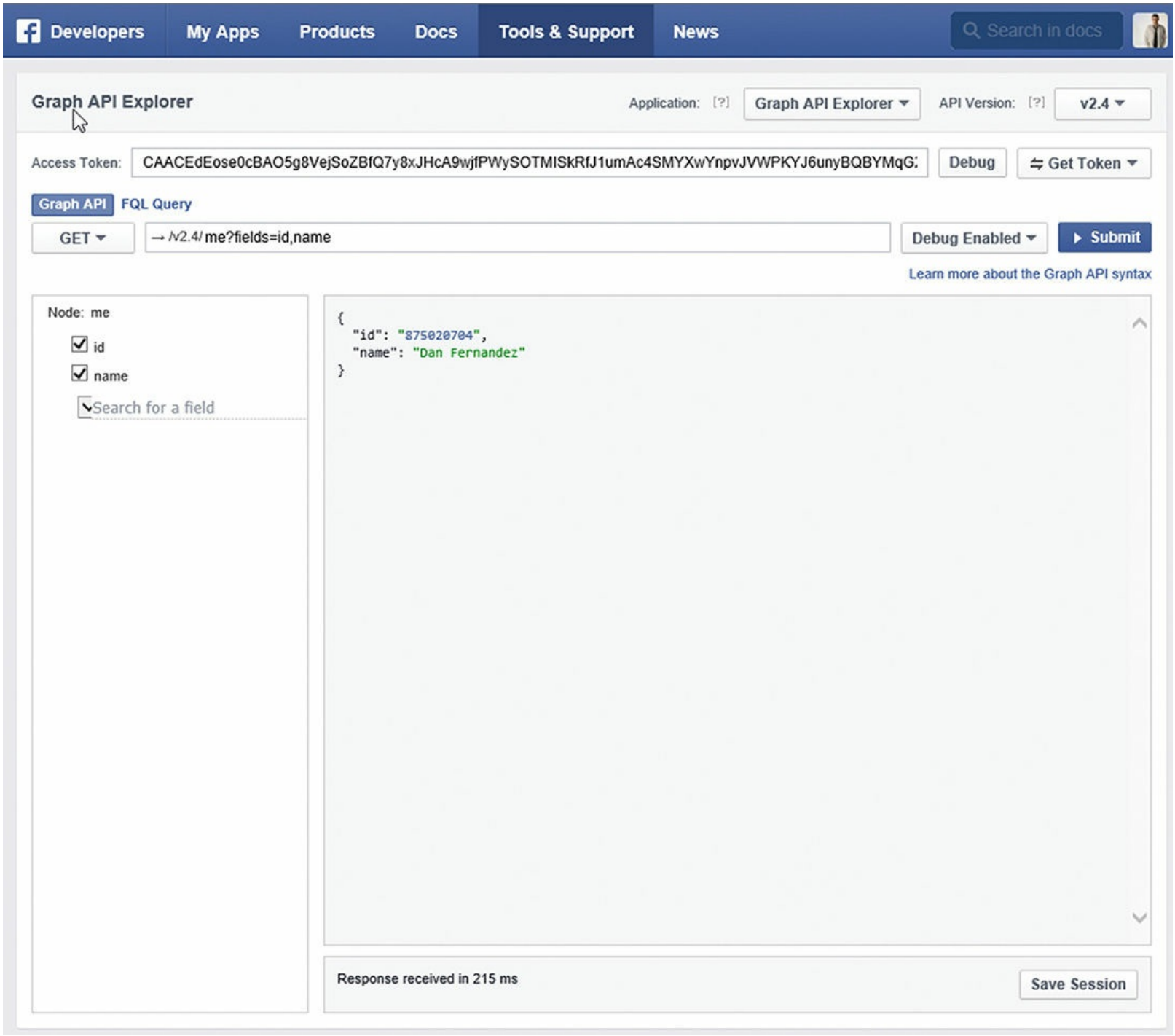

FIGURE A.2: Facebook's Graph API explorer enables you to test-drive their API from a browser

For ASP.NET Core 1.0 Web API developers, Ahoy!

At https://github.com/domaindrivendev/Ahoy is an ASP.NET Core 1.0 NuGet package that makes it easy to add Swagger support to your API. You can configure Ahoy! to use the XML comments from your classes to document the input parameters and return types used in your API. Once you do this, you'll get a beautiful documentation page as shown in Figure A.3, that includes a list of all your APIs. Each API includes a test harness page to test-drive the API.

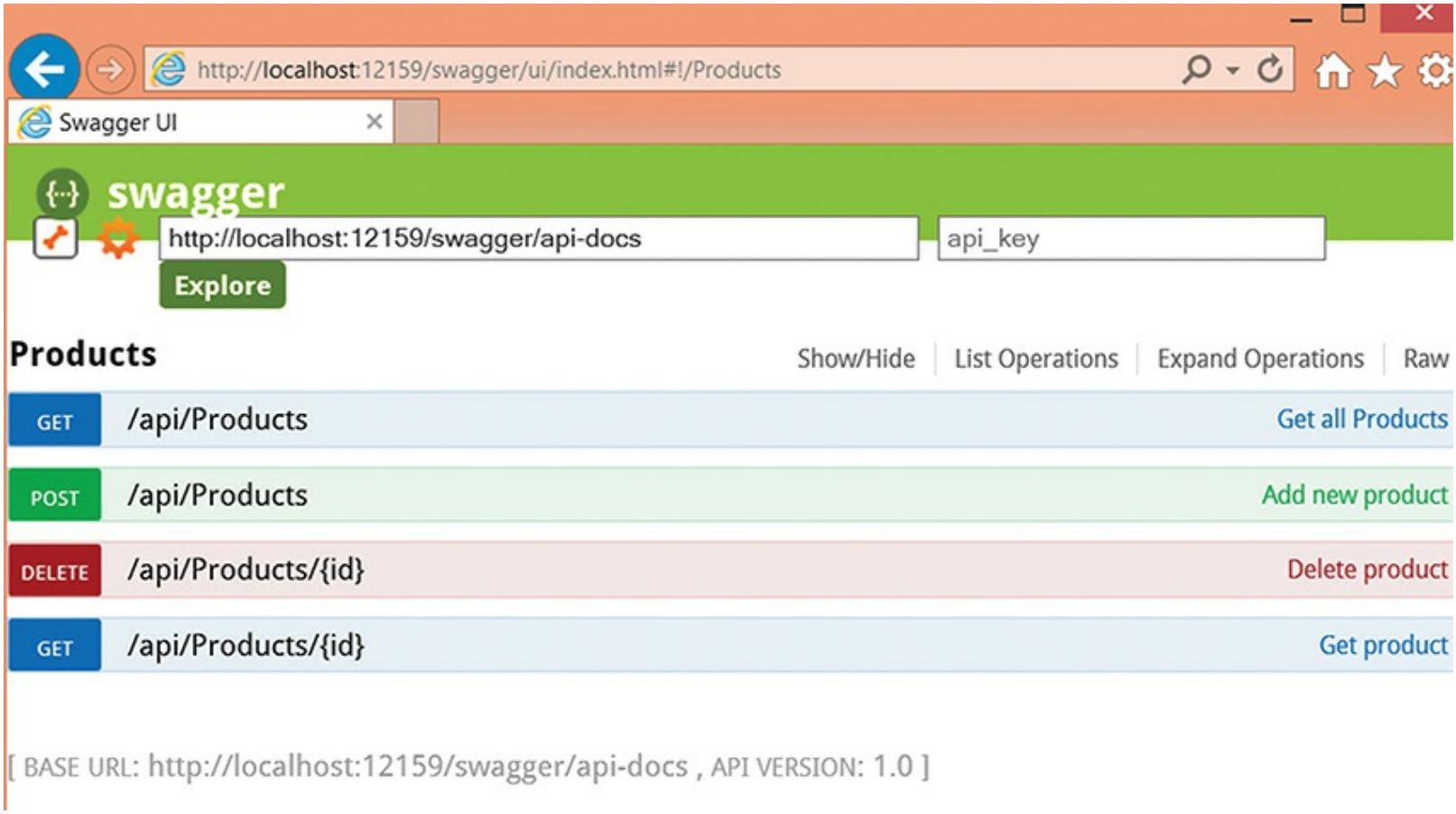

FIGURE A.3: Swagger shows the resources and URLs for the Products API

Consider Providing an "Uptime" Service

One of the most frustrating parts of working with distributed systems is tracking down where a particular service in a multiservice request is failing, or having delayed or limited results. For your most important microservices, you can help mitigate this by providing a service uptime microservice. As meta as that sounds, the idea is that you can provide a service dashboard page, much as you see major cloud vendors like Amazon, Microsoft, or Google provide for their services. This enables you to provide both automated service uptime listing each API and its status, as well as additional information your incident response team would add as they provide regular status updates on any issues your services are experiencing. The monitoring and diagnostics chapter dicusses the tools and services you can use to monitor your services.

Consider if OData is Right for You

OData, or the Open Data Protocol, is a standard, cross-platform way to provide queryable services via REST APIs. The advantage of OData is the developer productivity boost it provides to easily add deep querying semantics capability to your APIs with almost no work. Instead of building a series of REST services and inputs that include every possible way to filter or sort data, OData makes it trivial to add querying capabilities to your domain model. For example, in a single query, you can select the fields you want, like product id, name, and price; filter those products that are between $50 and $100 whose name starts with the letter "B"; and that are in the "Misc" product category. This powerful querying capability comes with some downsides. The first is that your data source and data model might not be optimized for this deep level of querying which can translate to unexpected performance issues. It's easier to understand the use cases and performance characteristics for a hand-crafted REST API. Another criticism of OData is that its overhead can add complexity to consumers of your

APIs, especially if OData client libraries aren't available for their target language or platform. There is no right choice here, but for API developers whose primary use case is rich querying capability, OData could be a great fit for you.

Index

A

A/B testing, 198
access to websites through the Internet, 52
ACS (Azure Container Service), 129–130, 132–133
ACS Resource Provider, 121–134
ADD, 60
adding
  content to containers, using volumes, 53–54
  dependencies to microservices, 111
agents, Apache Mesos, 148
aggregation, log aggregation, 217–218
Alpine, 93
announcements, service discovery, 151
anti-corruption layer, 83
Apache Mesos, 147–148
  agents, 148
  components, 148
  diverse workloads, 150
  frameworks, 148–149
  masters, 148
  Mesosphere DCOS (Data Center Operating System), 149
  service discovery, 150–152
API Gateway, 159–161
APIs
  authentication, OpenID, 270
  Batch APIs, ASP.NET, 270
  HTTP error codes, 268
  reliable actors API, 247
  reliable services API, 249–251
  RESTful APIs, 267–268
  schedulers, 141
application configuration changes, across different environments, 184–185
application data, logging from within containers, 222
application dependencies, project.json (ASP.NET), 257
application gateways, 121, 122, 159–161
Application Insights, 176, 227–231
application lifecycle, Azure Service Fabric, 251
application manifest, Azure Service Fabric, 241
application upgrades, Azure Service Fabric, 252–253

applications
  Azure Service Fabric, 240–241
    application manifest, 241
    custom applications, 242–243
    service manifest, 241–242
  decomposing, 74–75
    considerations for, 86–87
  designing
    bounded context, 75–76
    coarse-grained services, 70–72
    common versioning strategy, 77
    data collection, 81–83
    determining where to start, 70
    refactoring across boundaries, 75
    serialization, 78
    service design, 76–77
    service to service communication, 78
architects, skillsets and experience, 18
architecture
  flak.io e-commerce sample, 85–86
  microservices architecture, 2
ARM (Azure Resource Manager)
  creating environments, 177–179
  infrastructure as code, 126–127
  inside the box provisioning, 131–132
  multivendor provisioning, 135–136
  tracking deployments, with tags and labels, 179–181
ARM (Azure Resource Manager) templates, 36
  deploying
    to Azure, 134
    from version control, 135
ARM templating language, 127–128
  outputs, 130
  parameters, 128–129
  resources, 129–130
  variables, 129
artifact stores, 121
ASP.NET
  application dependencies, project.json, 257
  Async, 265–266
  Batch APIs, 270

choosing Docker images, 262–263
cloud-ready environment-based configurations, 258–259
command-line driven, 260
concurrency, 269
cross-platform apps, 256
cross-platform console applications, 261–262
cross-platform data and nonrelational data, 261
cross-platform web servers, 261
dependency injection (DI), 260–261
designing,
microservices, 269
for mobile clients, 270–271
environment variables, 114–115
front-end web development practices, 263–264
.NET Core, 256
NuGet, 257–258
OData, 274
open source stack, 256
OpenID, 271
RAD development, Roslyn, 261
REST services, 264–265
RESTful APIs, 267–268
stateless APIs, 266
Swagger, 272–273
unification of MVC and Web API, 260
uptime services, 273–274
Web API, return types, 266–267
ASP.NET Core 1.0, 255
changes, 109–110
Async, ASP.NET, 265–266
asynchronous messaging, 79–80
authentication
Docker, 91–92
OpenID, ASP.NET, 271
automated builds, Docker images, 98–99
automation
Azure environments, 174
microservices, 22–23
autonomous services, 4–5
autoscaling, schedulers, 141
availability sets, 139

Azure Service Fabric, 237
Azure
availability sets, 139
common log format, 221
containers, 34–35
creating environments, 173
with ARM (Azure Resource Manager), 177–179
automation, 174
immutable infrastructure, 173–174
infrastructure, 176
infrastructure as code, 174–175
private versus shared, 175–176
third-party configuration and deployment tools, 181
deploying ARM (Azure Resource Manager) templates, 134
deployment models, 42–44
Kubernetes, 147
load testing, 193–194
swarms, creating, 143–144
tracking deployments with tags and labels, 179–181
virtual machines, creating with Docker, 35–37
Azure Container Service (ACS), 129–130, 132–133
Azure Diagnostics, 222
Azure healing, schedulers, 140
Azure Portal, 36
Azure Resource Manager. *See* ARM (Azure Resource Manager)
Azure Resource Manager (ARM) templates, 36
Azure Service Fabric, 233, 234
application lifecycle, 251
application upgrades, 252–253
applications, 240–241
application manifest, 241
custom applications, 242–243
service manifest, 241–242
cluster management, 236–237
container integration, 243–244
fault domains, 238–240
Linux support, 234
programming model, 244
Quorum, 237
reliable actors API, 247
reliable services, reliable services API, 249–251

resource scheduling, 240
service discovery, 244
service updates, 251
stateless services, 244–245, 245–247
subsystems, 234–236
testability framework, 253–254
update domains, 238–240
virtual machine scale sets, 237

B

Bamboo, 205
base images, Docker, 92–95
Batch APIs, ASP.NET, 270
benefits of
DevOps, 22
microservices, 6
continuous innovation, 8–9
fault isolation, 12–13
independent deployments, 6–8
resource utilization, 9–10
scaling, 9–10
small teams, 12
technology diversity, 10–12
best practices, microservices, 19
automation, 22–23
Conway's Law, 20–21
DevOps, 21–22
encapsulation, 20
fault tolerance, 23–26
monitoring, 23
best-of-breed solutions, continuous delivery tools, 201
Big Bang Integration Testing, 192
bin packing, 138
bots, 185
boundaries, refactoring across, 75
bounded context, 75–76
bridge network, linking containers, 65
build agents, 201
build controllers, 201
Build Job, 205
build/test host, Docker, 90

bulkheads, 24

C

Calico, 165
canary testing, 196–197
cascading failures, 25
certificates, managing, in Docker, 91–92
challenges of
  manual deployments, 124
  microservices, 13–14
    complexity, 14
    data consistency, 15–16
    network congestion, 14–15
  scheduling, 136
changes to code, ASP.NET Core 1.0 changes, 109–110
Chaos Monkey, 199
choosing images for virtual machines, 40–41
Chronos, 149–150
CI (continuous integration), 171–172
circuit breakers, 24
CLAbot, 186
cloning samples, product catalog microservice, 106–107
cloud only, Docker, 91
cloud-config files, 132
cloud-ready environment-based configurations, ASP.NET, 258–259
cluster coordination, 156
cluster host environments, overview, 120
cluster management, 122
  Azure Service Fabric, 236–237
cluster resource manager, Azure Service Fabric, 235
cluster schedulers, autoscaling, 141
cluster scheduling, 138
cluster state store, 120
clusters
  application gateways, 159–161
  Docker Swarm, 141–142
  Swarm cluster, 141–142
    master nodes, 142
CMD, 60
coarse-grained services, 70–72
  decomposing, 72

defining services and interfaces, 73–74
microservices, 72–73
code analysis with SonarQube, 190
code samples, flak.io microservices sample code, 4
coded UI testing, 192–193
cohesion, 74–75
collaboration, DevOps, 170–171
command-line driven, ASP.NET, 260
commands
ADD, 60
CMD, 60
COPY, 60
ENTRYPOINT, 60
ENV, 60
EXPOSE, 60
FROM, 59
MAINTAINER, 59
ONBUILD, 60
RUN, 59
USER, 60
VOLUME, 60
WORKDIR, 60
committing images, 56–61
common log format, 219–221
common versioning strategy, 77
communication, service to service communication, 78
communication subsystem, Azure Service Fabric, 235
complexity, microservices, 14
concurrency, ASP.NET, 269
connecting
to Docker hosts, 105
to swarms, Docker Swarm, 144
to virtual machines
with SSH and Git Bash on Mac OS X, 46
with SSH and Git Bash on Windows, 44
connectivity issues, unable to connect to the host, Docker, 116–117
consistency, infrastructure as code, 125
console applications, cross-platform console applications (ASP.NET), 261–262
constraints, placement constraints, 242
Consul, 158
consumer-driven contract testing, service dependencies, 187–189

container events, overview, 214
container ids, deleting containers, 54
container integration, Azure Service Fabric, 243–244
container linking, 114
container logs, viewing, 63–64
container networking, 64
containers, 29–30
  adding content to with volumes, 53–54
  Azure, 34–35
  creating new, 61
  deleting, 60
    with container ids, 54
  dependencies, 114
  Docker, 30, 34–35
  Docker issues, containers that won't start, 117
  enabling live reload, 107–108
  images, updating and committing, 56–61
  linking, 65
    Docker, 114
  logging application data, 222
  monitoring, 212–213, 216
    Docker Remote API, 215–216
    Docker runtime metrics, 213–215
  noisy neighbor problem, 32–34
  versus processes, 30–32
  reactive security, 34
  resource scheduling, 240
  schedulers, 139
  versus virtual machines, 30–32
continuous delivery, 171–172, 200
  deploying microservices, 182–184
continuous delivery tools
  best-of-breed solutions, 201
  considerations when choosing, 203–204
  extensibility, 202–203
  hybrid pipelines, 202
  on-premises or hosted service, 201
  on-premises or hosted tools, 200
continuous deployment, 171–172
continuous innovation, microservices, 8–9
continuous integration (CI), 185

code analysis with SonarQube, 190
improving quality through pull request validation, 185–186
public third-party services, 190
testing service dependencies with consumer-driven contract testing, 187–189
unit testing, 186–187
website performance, 190–191
Conway's Law, 20–21
COPY, 60
CoreOS, etcd, 158
correlation ID, monitoring services, 218
coupling, 74
cross-platform apps, ASP.NET, 256
cross-platform console applications, ASP.NET, 261–262
cross-platform data and nonrelational data, ASP.NET, 261
cross-platform web servers, ASP.NET, 261
culture, DevOps, 170–171
collaboration, 170–171
demystifying deployments, 170
no blame rule, 170
custom applications, Azure Service Fabric, 242–243

D

data collection, 81–83
data consistency, microservices, 15–16
DDD (Domain Driven Design), 6, 75
debugging Docker issues
containers that won't start, 117
diagnosing running containers, 118
unable to connect to the host, 116–117
decomposing
applications, 74–75
considerations for, 86–87
coarse-grained services, 72
defining services and interfaces, 73–74
microservices, 72–73
dedicated cluster nodes, 161
dedicated gateways, 161
deleting containers, 60
with container ids, 54
demystifying deployments, DevOps, 170
dependencies

adding to microservices, 111
containers, 114
schedulers, 139
dependency injection (DI), 260–261
deploying
ARM (Azure Resource Manager) templates
to Azure, 134
from version control, 135
microservices, with continuous delivery, 182–184
to staging, 195
deployment models, 42–44
deployments
demystifying deployments, DevOps, 170
independent deployments, microservices, 6–8
manual deployments, challenges of, 124
sharing, 157
tracking changes with tags and labels, 179–181
designing
applications
bounded context, 75–76
coarse-grained services, 70–72
common versioning strategy, 77
data collection, 81–83
determining where to start, 70
refactoring across boundaries, 75
serialization, 78
service design, 76–77
service to service communication, 78
microservices, ASP.NET, 269
for mobile clients, ASP.NET, 270–271
developer configurations, Docker
cloud only, 91
local and cloud, 91
local development, 90
developer tools, installing, 102
development machines, installing (Docker), 37
DevOps
continuous delivery, 200
on-premises or hosted tools, 200
culture, 170–171
collaboration, 170–171

demystifying deployments, 170
no blame rule, 170
support, 170
Dockerizing, 181–182
microservices, 21–22
overview, 167–168, 169
testing, 192
A/B testing, 198
canary testing, 196–197
coded UI testing, 192–193
Docker stress testing, 195
fault tolerance testing, 198–199
integration testing, 192
load and stress testing, 193
manual/exploratory testing, 196
resiliency testing, 198–199
DI (dependency injection), 260–261
diagnosing running containers, 118
directories, creating with Docker, 54–55
discovery backend, Docker Swarm, 142
diverse workloads, Apache Mesos, 150
DNS, service discovery, 157
DNS protocol integrations, service registry, 156
Docker, XVIII
authentication, 91–92
build/test host, 90
container linking, 114
containers, 30, 34–35
creating Azure virtual machines, 35–37
developer configurations
cloud only, 91
local and cloud, 91
local development, 90
directories, creating, 54–55
images, 49–50
automated builds, 98–99
building a hierarchy, 95–98
choosing base images, 92–95
choosing with ASP.NET, 262–263
sharing, 99–100
tags, 99

installing
on Azure virtual machines, 36
on development machines, 37
local development, 89, 103
settings, 103–104
monitoring recommended solutions, 232
networking features, overlay networks, 163–164
product validation, 90
runtime metrics, 213–215
stress testing, 195
TLS (Transport Layer Security), 144
tracking deployments with labels, 180–181
verifying installation, 47–48
Docker Bench, 100
Docker Cloud, 206–207
Docker Compose, 102, 112
smart restart, 115–116
Docker Engine, 101
Docker Exec, 118
Docker for Mac, 102
Docker for Windows, 102
Docker hosts, connecting to, 105
Docker Hub, 48, 93–94
images, 94–95
Docker issues, debugging
containers that won't start, 117
diagnosing running containers, 118
unable to connect to the host, 116–117
Docker Kitematic, 101
docker logs, viewing, 63–64
Docker Machine, 101
docker ps, 50–52
docker pull, 50
docker pull nginx, 49
Docker Quickstart Terminal, 104–105
Docker Remote API, 213, 215–216
docker run, 50–52
Docker run command, 181
docker search nginx, 49
Docker Swarm, 141–142
connecting to swarms, 144

creating swarms, on Azure, 143–144
discovery backend, 142
master nodes, 142
strategies, 142–143
swarm filters, 143
Docker Swarm Cluster Template, 143–144
Docker tools, installing, 101–102
Docker Trusted Registry (DTR), 100
Dockerfiles, 98
building images, 59–60
dockerhostkey, 38
dockerhostkey.pub, 38
Dockerizing, 181–182
Domain Driven Design (DDD), 6, 75
DTR (Docker Trusted Registry), 100

E

efficiency, scheduling, 137
elasticsearch, 105
encapsulation, microservices, 20
Enterprise Service Bus (ESB), 21
ENTRYPOINT, 60
ENV, 60
environment variables, 68
ASP.NET, 114–115
environments
application configuration changes, 184–185
Azure, 173
automation, 174
creating with ARM (Azure Resource Manager), 177–179
immutable infrastructure, 173–174
infrastructure, 176
infrastructure as code, 174–175
private versus shared, 175–176
third-party configuration and deployment tools, 181
removing, 125
updating, infrastructure as code, 124–125
ESB (Enterprise Service Bus), 21
etcd, 158
Eureka, 158
exception processing, 198

expectations, consumer-driven contract testing, 187
experience, architects, 18
exploratory testing, 196
EXPOSE, 60
expressions, templates, 128
extensibility, continuous delivery tools, 202–203
extensions, virtual machines, 37

F

failover manager service, Azure Service Fabric, 235
failures, cascading failures, 25
fault domains, Azure Service Fabric, 238–240
fault isolation, microservices, 12–13
fault tolerance, microservices, 23–26
fault tolerance testing, 198–199
features, service registry, 155
federation subsystem, Azure Service Fabric, 235
files
  dockerhostkey, 38
  dockerhostkey.pub, 38
filters, swarm filters, Docker Swarm, 143
fine-grained SOA. *See* microservices architecture
flak.io e-commerce sample, 83
  architecture overview, 85–86
  requirements, 84–85
flak.io microservices sample code, 4
Flannel, 164–165
format of logs, 219–221
frameworks
  Apache Mesos, 148–149
  testability framework, Azure Service Fabric, 253–254
FROM, 59
front-end web development practices, ASP. NET and Visual Studio 2015, 263–264
functions, templates, 128

G

Gatekeeper, 196–197
gateways, application gateways, 159–161
generating SSH public keys
  on Mac OS X, 39
  on Windows, 37–39

Git Bash, 37
  connecting virtual machines, on Mac OS X, 46
Git command line, 102
Git Credential Manager, 102
Google, Page Speed Insights API, 190–191
Grunt, 111
  phantomas task, 190–191
Gulp, 111

H

HAProxy, 160
HashiCorp
  Consul, 158
  Terraform, 135–136
health checks, service registry, 156
health subsystem, Azure Service Fabric, 234–235
hierarchies, images hierarchies (Docker), 95–98
high availability
  schedulers, 140
  service registry, 155
horizontal scaling, 32
host machines, monitoring, 210–211
host nodes, 120
host-agnostic, 112
hosted infrastructure, continuous delivery tools, 200
hosting subsystem, Azure Service Fabric, 235
HTTP error codes, APIs, 268
hybrid pipelines, 202
hypervisor virtualization technologies, 30–31

I

-i (identify file parameter), 45
IaaS (Infrastructure as a Service), 119
identify file parameter (-i), 45
image layering, 61–63
image registry, 121
image repositories, Docker, 48
images
  choosing
    ASP.NET Docker images, 262–263
    for virtual machines, 40–41

Docker, 49–50
automated builds, 98–99
building a hierarchy, 95–98
choosing base images, 92–95
sharing, 99–100
tags, 99
managing, 100–101
updating and committing, 56–61
immutable infrastructure, environments (Azure), 173–174
independent deployments, microservices, 6–8
Information blade, 40–41
infrastructure
Azure environments, 176
hosted infrastructure, 200
immutable infrastructure, 173–174
Infrastructure as a Service. *See* IaaS (Infrastructure as a Service)
infrastructure as code, 123–124
ARM (Azure Resource Manager), 126–127
Azure, 174–175
consistency, 125
removing, old environments, 125
test-driven infrastructure, 126
tracking changes, 125
updating environments, 124–125
innovation, continuous innovation (microservices), 8–9
inside the box provisioning, 123
ARM (Azure Resource Manager), 131–132
installing
developer tools, 102
Docker, on Azure virtual machines, 36
Docker tools, 101–102
OSX utilities, 103
Windows utilities, 102–103
integrated solutions, continuous delivery tools, 201–202
integration
container integration, Azure Service Fabric, 243–244
continuous integration, 185
code analysis with SonarQube, 190
improving quality through pull request validation, 185–186
public third-party services, 190
testing service dependencies with consumer-driven contract testing, 187–189

unit testing, 186–187
website performance, 190–191
microservices, 17
integration testsstress testing, 76–77
interfaces, defining for coarse-grained services, 73–74
inter-service communications, 80
isolation, scheduling, 137

J

javascript task runners, 111–112
Jenkins, 205
JSON, parameters, 128–129
JSON-based documents, ARM templating language, 127–128

K

Kanban Tool, 202–203
key vault, 176
kubelets, 145
Kubernetes, 144–145
Azure, 147
components, 145, 147
labels, 146
names, 147
namespaces, 147
pods, 145–146
replication controllers, 146
selectors, 146
services, 146
volumes, 146–147

L

labels
Kubernetes, 146
tracking deployments, 179–181
Docker, 180–181
languages, ARM templating language, 127–128
layering, images, 61–63
linked templates, 131
linking containers, 65
Docker, 114
Linux
Azure Service Fabric, 234

Windows and, 35
Linux CPU, 225
Linux diagnostics event, 223–225
Linux VirtualBox Path, 108–109
LinuxDisk, 225–226
LinuxMemory, 226
ListAsync(-) method, 265
live reload, enabling, 107–108
load balancers, 153, 160, 176
load testing, 193
with Azure and Visual Studio, 193–194
local and cloud, Docker, 91
local development
Docker, 89, 90, 103
settings, 103–104
installing Docker tools, 101–102
local Docker hosts, starting, 104–105
settings, 103–104
local Docker hosts, starting, 104–105
log aggregation, 217–218
logging, application data, from within containers, 222
logs
common log format, 219–221
considerations for, 221–222
Logstash, 220
lookup API, service registry, 155
lookups, service discovery, 151
ls command, 109

M

Mac computers, cloning samples, 106
Mac OS X
connecting virtual machines with SSH and Git Bash, 46
generating SSH public keys, 39
MAINTAINER, 59
management services, 120
management subsystem, Azure Service Fabric, 234
managing images, 100–101
manual deployments, challenges of, 124
manual testing, 196
as a service, 196

Marathon, 149
master nodes, 142
masters, Apache Mesos, 148
Mean Time to Repair (MTTR), 169
Mean Time to Resolution (MTTR), 7
Meso-DNS, 157
Mesosphere DCOS (Data Center Operating System), 149
  Chronos, 149–150
  Marathon, 149
microservices, XVII–XVIII, 2–4
  autonomous services, 4–5
  benefits of, 6
    continuous innovation, 8–9
    fault isolation, 12–13
    independent deployments, 6–8
    resource utilization, 9–10
    scaling, 9–10
    small teams, 12
    technology diversity, 10–12
  best practices, 19
    automation, 22–23
    Conway's Law, 20–21
    DevOps, 21–22
    encapsulation, 20
    fault tolerance, 23–26
    monitoring, 23
  challenges of, 13–14
    complexity, 14
    data consistency, 15–16
    network congestion, 14–15
  changing monoliths to, 80–81
  cross-grained services, 70–72
  decomposing coarse-grained services, 72–73
  defining service boundaries, 6
  deploying with continuous delivery, 182–184
  designing, ASP.NET, 269
  integration, 17
  monitoring, 18
  preparing for production, 110
    adding dependencies, 111
    javascript task runners, 111–112

optimizing source code, 111
product catalog microservice, 105
cloning samples, 106–107
enabling live reload, 107–108
Linux VirtualBox Path, 108–109
volumes, 108
retry patterns, 26
routing, 17–18
service dependencies, 25
service discovery, 17–18
single responsibility principle, 5–6
skillsets for architects, 18
SLA (service level agreement), 19
small services, 5
testing, 16–17, 112
time to live (TTL), 16
versioning, 17
microservices architecture, 2
monitoring, 209–210
Application Insights, 227–231
Azure Diagnostics, 222
containers, 212–213, 216
Docker Remote API, 215–216
Docker runtime metrics, 213–215
host machines, 210–211
microservices, 18, 23
OMS (Operations Management Suite), 231–232
recommended solutions by Docker, 232
services, 216–217
common log format, 219–221
correlation ID, 218
log aggregation, 217–218
operational consistency, 218–219
solutions, 222
syslog drivers, 227
monoliths, 71
changing to microservices, 80–81
partitioning, 82
refactoring, 81
MTTR (Mean Time to Repair), 169
MTTR (Mean Time to Resolution), 7

multivendor provisioning, ARM (Azure Resource Manager), 135–136

N

names, Kubernetes, 147
namespaces, Kubernetes, 147
.NET Core
  ASP.NET, 256
  support, 256–257
Netflix, Prana, 219
network congestion, microservices, 14–15
networking
  container networking, 64
  overlay networks, 65–67
networks, overlay networks, 161–163
  Docker networking feature, 163–164
  Flannel, 164–165
  Project Calico, 165
  Weave Net, 164
NGINX, 160
  Docker, images, 49–50
no blame rule, DevOps, 170
Node Docker image, 94
nodes, master nodes, 142
noisy neighbor problem, containers, 32–34
notifications, service registry, 156
NuGet, 109–110
  ASP.NET, 257–258

O

OData, 274
OMS (Operations Management Suite), 231–232
ONBUILD, 60
on-premises tools, 200
open source stack, ASP.NET, 256
OpenID, authentication (ASP.NET), 271
operational consistency, monitoring services, 218–219
Operations Management Suite (OMS), 231–232
optimizing source code, microservices, 111
Oracle VirtualBox, 101
orchestration, 121–122
  Apache Mesos. *See* Apache Mesos

cluster management, 122
Docker Swarm. *See* Docker Swarm
discovery backend, 142
master nodes, 142
Kubernetes. *See* Kubernetes
master nodes, 142
provisioning, 121, 123
schedulers. *See* schedulers
scheduling, 122, 136
challenges of, 136
efficiency, 137
isolation, 137
performance, 138
scalability, 137
solutions for, 138
service discovery, 150–152
service lookup, 153–155
service registration, 152–153
organizing images (Docker), with tags, 99
OSX utilities, installing, 103
outputs, ARM templating language, 130
outside the box provisioning, 123
overlay networks, 65–67, 122–123, 161–163
Docker networking feature, 163–164
Flannel, 164–165
Project Calico, 165
Weave Net, 164

P

-p (port parameter), 45
PaaS (Platform as a Service), 119
Pact, 16
consumer-driven contract testing, 189
PageSpeed task, 190–191
parameters
ARM templating language, 128–129
-i (identify file parameter), 45
linked templates, 131
-p (port parameter), 45
partitioning, 82
monoliths, 82

partitions, stateless services, 245
patterns
  retry patterns, microservices, 26
  sidecar patterns, 219
peer gateway request routing, 161
performance
  scheduling, 138
  website performance, 190–191
pets and cattle metaphor, 125
phantomas Grunt task, 190–191
placement constraints, 242
Platform as a Service (PaaS), 119
pods, Kubernetes, 145–146
port parameter (-p), 45
Prana, 219
preparing for production, microservices, 110
  adding dependencies, 111
  javascript task runners, 111–112
  optimizing source code, 111
prioritization, 82
private environments versus shared environments, 175–176
processes versus containers, 30–32
product catalog microservice, 105
  cloning samples, 106–107
  enabling, live reload, 107–108
  Linux VirtualBox Path, 108–109
  volumes, 108
product validation, Docker, 90
production, testing in, 196
programmable infrastructure, 123–124
programming model, Azure Service Fabric, 244
Project Calico, 165
project.json, ASP.NET, 257
provisioning, 121–122, 123
  inside the box provisioning, ARM (Azure Resource Manager), 131–132
  multivendor provisioning, ARM (Azure Resource Manager), 135–136
proxies, routing proxies, 155
public third-party services, continuous integration, 190
pull request validation, improving quality through, 185–186
Putty client, 102

Q

QA environments
  deploying to staging, 195
  testing, 192
Quay Enterprise, 100
Quorum, Azure Service Fabric, 237

R

RAD development, with Roslyn, ASP.NET, 261
RainforestQA, 196
random approach, 138
rating services, containers, 32–34
reactive security, 34
reallocation, schedulers, 140
refactoring, monoliths, 81
refactoring across boundaries, 75
registry, service discovery, 151
reliability subsystem, Azure Service Fabric, 235
reliable actors API, Azure Service Fabric, 247
reliable services API, Azure Service Fabric, 249–251
removing old environments, infrastructure as code, 125
replicas, stateless services, 245–246
replication
  schedulers, 139–140
  stateless services, 246
replication controllers, Kubernetes, 146
requirements, flak.io e-commerce sample, 84–85
resiliency testing, 198–199
resource scheduling, Azure Service Fabric, 240
resource utilization, microservices, 9–10
resources, ARM templating language, 129–130
REST services, ASP.NET, 264–265
RESTful APIs, ASP.NET, 267–268
retry, 24
retry patterns, microservices, 26
return types, ASP.NET Web API, 266–267
Robinson, Ian, 188
rolling updates, schedulers, 140
Roslyn, RAD development, ASP.NET, 261
routing, microservices, 17–18
routing proxies, 155
RUN, 59

runtime metrics, Docker, 213–215

S

scalability
  scheduling, 137
  service registry, 155
scaling
  autoscaling, 141
  microservices, 9–10
  virtual machines, 31
scaling out, 32
scaling up, 32
schedulers
  API, 141
  autoscaling, 141
  availability sets, 139
  Azure healing, 140
  bin packing, 138
  constraints, 139
  dependencies, 139
  high availability, 140
  random approach, 138
  reallocation, 140
  replication, 139–140
  rolling updates, 140
  spread approach, 138
scheduling, 122, 136
  challenges of, 136
  efficiency, 137
  isolation, 137
  performance, 138
  scalability, 137
  solutions for, 138
security, reactive security, 34
security tools, Docker Bench, 101
selectors, Kubernetes, 146
Selenium, 192–193
serialization, 78
  asynchronous messaging, 79–80
  synchronous request/response, 78–79
service, defining for coarse-grained services, 73–74

service announcements, 152–153
  service registry, 155
service boundaries, microservices, 6
service decomposition, coarse-grained services, 72
  defining services and interfaces, 73–74
  microservices, 72–73
service dependencies
  microservices, 25
  testing with consumer-driven contract testing, 187–189
service design, 76–77
service discovery, 122
  Apache Mesos, 150–152
  Azure Service Fabric, 244
  Consul, 158
  DNS, 157
  etcd, 158
  Eureka, 158
  microservices, 17–18
  Zookeeper, 158
service discovery store, 120
Service Fabric. *See* Azure Service Fabric
Service Fabric Replicator, 235
service level agreements. *See* SLA (service level agreement)
service lookup, 153–155
service manifest, Azure Service Fabric, 241–242
service registration, 152–153
service registry, 155
  DNS protocol integrations, 156
  features, 155
  health checks, 156
  high availability, 155
  lookup API, 155
  notifications, 156
  scalability, 155
  service announcements, 155
service to service communication, 78
service updates, Azure Service Fabric, 251
services
  Kubernetes, 146
  manual testing, 196
  monitoring, 216–217

common log format, 219–221
correlation ID, 218
log aggregation, 217–218
operational consistency, 218–219
set OSX environment variables, 105
set Windows environment variables, 105
shared environments, versus private environments, 175–176
sharing
deployments, 157
images (Docker), 99–100
sidecar patterns, 219
single responsibility principle, 5–6
SkyDNS, 157
SLA (service level agreement), microservices, 19
Slack, 202–203
small services, microservices, 5
small teams, microservices, 12
smart restart, Docker Compose, 115–116
solutions, monitoring, 222
solutions for, scheduling, 138
SonarQube, code analysis, 190
Spotify, 195
spread approach, 138
SSH, connecting virtual machines, on Mac OS X, 46
SSH public keys, generating
on Mac OS X, 39
on Windows, 37–39
staging, deploying to, 195
starting local Docker hosts, 104–105
stateless APIs, ASP.NET, 266
stateless services, Azure Service Fabric, 244–247
STDERR, 222
STDOUT, 222
storage, 176
stress testing, 193
Docker, 195
subsystems, Azure Service Fabric, 234–236
support, .NET Core, 256–257
Swagger, 264–265
ASP.NET, 272–273
Swarm cluster, 141–142

master nodes, 142
swarm filters, Docker Swarm, 143
swarm strategies, Docker Swarm, 142–143
swarms
connecting to, Docker Swarm, 144
creating with Docker, on Azure, 143–144
synchronous request/response, 78–79
syslog drivers, 227

T

tags
images, Docker, 99
tracking deployments, 179–181
teams, microservices, 12
technical debt, 171
technologies
application gateways, 159–161
Consul, service discovery, 158
DNS, service discovery, 157
etcd, 158
Eureka, 158
Zookeeper, 158
technology diversity, microservices, 10–12
templates
ARM (Azure Resource Manager) templates, 36
deploying from version control, 135
deploying to Azure, 134
Docker Swarm Cluster Template, 143–144
expressions, 128
functions, 128
linked templates, 131
Terraform, 135–136
testability framework, Azure Service Fabric, 253–254
test-driven infrastructure, 126
testing
DevOps, 192
A/B testing, 198
canary testing, 196–197
coded UI testing, 192–193
Docker stress testing, 195
fault tolerance testing, 198–199

integration tests, 192
load and stress testing, 193
manual/exploratory testing, 196
resiliency testing, 198–199
integration tests, 76–77
load testing with Azure and Visual Studio, 193–194
manual testing, as a service, 196
microservices, 16–17, 112
in production, 196
service dependencies with consumer-driven contract testing, 187–189
unit testing, 186–187
user acceptance testing, 195
third-party configuration and deployment tools, Azure, 181
third-party services, continuous integration, 190
time to live (TTL), 16
timeouts, 24
TLS (Transport Layer Security), Docker, 144
tools
developer tools, installing, 102
Docker Compose, 112
Docker tools, installing, 101–102
Toxiproxy, 199
tracking changes, infrastructure as code, 125
tracking deployments, with tags and labels, 179–181
Transport Layer Security (TLS), Docker, 144
transport subsystem, Azure Service Fabric, 235
tribal knowledge, 170–171
TTL (time to live), 16
turns, Azure Service Fabric, 248
Tutum, 206–207

U

unit testing, 186–187
update domains, Azure Service Fabric, 238–240
updating
environments, infrastructure as code, 124–125
images, 56–61
upgrades, application upgrades, Azure Service Fabric, 252–253
uptime services, ASP.NET, 273–274
USER, 60
user acceptance testing, 195

UserVoice, 202–203

V

variables
  ARM templating language, 129
  environment variables, 68
    ASP.NET and, 114–115
  set OSX environment variables, 105
  set Windows environment variables, 105
verifying Docker installation, 47–48
version control, deploying, ARM (Azure Resource Manager) templates, 135
versioning
  common versioning strategy, 77
  microservices, 17
vertical scaling, 32
viewing, container logs, 63–64
virtual machine scale sets, 237
virtual machines
  choosing images, 40–41
  connecting
    with SSH and Git Bash on Mac OS X, 46
    with SSH and Git Bash on Windows, 44
  versus containers, 30–32
  creating with Docker, 35–37
  extensions, 37
  installing, Docker, 36
  scaling, 31
virtual networks, 176
virtual private networks (VPNs), 176
VirtualBox, 105
Visual Studio, load testing, 193–194
Visual Studio 2015, front-end web development practices, 263–264
Visual Studio Code, 102, 112–113
Visual Studio Team Services (VSTS), 201, 205
VOLUME, 60
volumes
  adding content to containers, 53–54
  Kubernetes, 146–147
  product catalog microservice, 108
VPNs (virtual private networks), 176
VSTS (Visual Studio Team Services), 201

W

Weave Net, 164
Web API, ASP.NET, return types, 266–267
web servers, cross-platform web servers, ASP.NET, 261
website performance, continuous integration, 190–191
Windows
  cloning samples, 106
  connecting virtual machines, with SSH and Git Bash on Windows, 44
  diagnostics extension, 223
  generating, SSH public keys, 37–39
  Linux and, 35
Windows Server 2016 containers, 179
Windows Server Containers, 179
Windows utilities, installing, 102–103
WORKDIR, 60
workloads, Apache Mesos, 150

X-Y

XML, ASP.NET, 267

Z

Zookeeper, 158
Zuul, 196–197

Microservices with DOCKER on Microsoft® Azure

This book is part of InformIT's exciting new Content Update Program, which provides automatic content updates for major technology improvements!

- As significant updates are made to Docker and Azure, sections of this book will be updated or new sections will be added to match the updates to the technologies.
- The updates will be delivered to you via a free Web Edition of this book, which can be accessed with any Internet connection.
- This means your purchase is protected from immediately outdated information!

For more information on InformIT's Content Update program, see the inside back cover or go to **informit.com/cup**

If you have additional questions, please email our Customer Service department at ***informit@custhelp.com***.

Microservices with DOCKER on Microsoft® Azure

Instructions to access your free copy of *Microservices with Docker on Microsoft Azure* Web Edition as part of the Content Update Program:

If you purchased your book from informit.com, your free Web Edition can be found under the **Digital Purchases** tab on your Account page.

If you have not registered your book, follow these steps:

1. Go to **informit.com/register**.
2. Sign in or create a new account.
3. Enter ISBN: **9780672337499**.
4. Answer the questions as proof of purchase.
5. Click on the **"Digital Purchases"** tab on your Account page to access your free Web Edition.

More About the Content Update Program...

InformIT will be updating the *Microservices with Docker on Microsoft Azure* Web Edition periodically, when significant updates are made to Docker and Azure.

To receive an email alerting you of the changes each time the *Microservices with Docker on Microsoft Azure* Web Edition has been updated, opt-in to receive emails from us when you create your account, or sign up for promotional emails at informit.com/newsletters. The email address you use to sign up for promotional emails must be the same email address used for your informit.com account in order to receive the alerts.

If you do not want to receive InformIT emails, you can check your informit.com account for the latest Web Edition.

When a new edition of this book is published, no further updates will be added to this book's Web Edition. However, you will continue to have access to your current Web Edition with its existing updates.

The Web Edition can be used on tablets that use modern mobile browsers. Simply log into your informit.com account and access the Web Edition from the **"Digital Purchases"** tab.

For more information about the Content Update Program, visit **informit.com/cup** or email our Customer Service department at **informit@custhelp.com**.

REGISTER YOUR PRODUCT at informit.com/register

Access Additional Benefits and SAVE 35% on Your Next Purchase

- Download available product updates.
- Access bonus material when applicable.
- Receive exclusive offers on new editions and related products. (Just check the box to hear from us when setting up your account.)
- Get a coupon for 35% for your next purchase, valid for 30 days. Your code will be available in your InformIT cart. (You will also find it in the Manage Codes section of your account page.)

Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.

InformIT.com–The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com you can

- Shop our books, eBooks, software, and video training.
- Take advantage of our special offers and promotions (informit.com/promotions).
- Sign up for special offers and content newsletters (informit.com/newsletters).
- Read free articles and blogs by information technology experts.
- Access thousands of free chapters and video lessons.

Connect with InformIT–Visit informit.com/community

Learn about InformIT community events and programs.

informIT.com
the trusted technology learning source

Addison-Wesley • Cisco Press • IBM Press • Microsoft Press • Pearson IT Certification • Prentice Hall • Que • Sams • VMware Press

ALWAYS LEARNING

PEARSON

Code Snippets

```
docker-machine_linux-amd64 create -d azure
      --azure-subscription-
id="<subscriptionID>"
      --azure-subscription-cert="mycert.pem"
   machine-name
```

```
$ ssh -i ./dockerhostkey dockeradmin@dockerhost-3udvgzn4.cloudapp.net
-p 22
```

```
$ ssh -i ./dockerhostkey dockeradmin@dockerhost-3udvgzn4.cloudapp.net
-p 22
```

```
docker run --name webcontainer -p 80:80 -d nginx
```

```
<html>
    <head>
    </head>
    <body>
            This is a Hello from a website in a container!
    </body>
</html>
```

```
docker run -name webcontainer -v /home/src:/usr/share/nginx/html:ro
-p 80:80 -d nginx
```

```
-v /home/src:/usr/share/nginx/html:ro
```

```
docker run -t -i nginx /bin/bash.
```

```
docker commit -m "updates applied" -a "Boris Scholl" 67337e2dbcbb
bscholl/nginx:v1
```

```
#Simple WebSite
FROM nginx
MAINTAINER Boris Scholl <bscholl@flak.io>
COPY web /usr/share/nginx/html
EXPOSE 80
```

```
docker run --name webcontainer -d -p 80:80 customnginx
```

```
MAINTAINER Boris Scholl <bscholl@flak.io>
COPY web /usr/share/nginx/html
EXPOSE 80
```

```
172.17.0.1 - - [12/Dec/2015:17:16:11 +0000] "GET / HTTP/1.1" 200 95
"-" "curl/7.38.0" "-"
172.17.0.1 - - [12/Dec/2015:17:51:55 +0000] "GET / HTTP/1.1" 200 95
"-" "curl/7.38.0" "-"
```

```
docker logs --follow webcontainer.
```

```
docker run -name webcontainer -net=host -d - p 80:80 customnginx
```

```
docker run --name webcontainer -d -p 80:80 customnginx
```

```
docker run --name webcontainer2 --link webcontainer:weblink -d -p
85:80 customnginx
```

```
docker run --name webcontainer2 --link webcontainer:weblink -d -p
85:80 -e SQL_CONNECTION='staging' customnginx
```

```
set DOCKER_CERT_PATH=c:\users\<username>\.docker\machine\machines\
finance-dev
```

```
export DOCKER_CERT_PATH=~/.docker/machine/machines/finance-dev
```

```
Set DOCKER_CERT_PATH=C:\Users\danielfe\.docker\machine\machines\
default
Set DOCKER_TLS_VERIFY=1
Set DOCKER_HOST=tcp://192.168.99.100:2376
```

```
export DOCKER_CERT_PATH=~/.docker/machine/machines/default
export DOCKER_TLS_VERIFY=1
export DOCKER_HOST=tcp://192.168.99.100:2376
```

```
cd ~
mkdir DockerBook
cd DockerBook
git clone https://github.com/flakio/catalog
```

```
cd c:\Users\<username>\Documents\
mkdir DockerBook
cd DockerBook
Git clone https://github.com/flakio/catalog
```

```
docker run -d -p 9200:9200 elasticsearch
```

```
docker run -d -t -p 80:80 -e “server.urls=http://*:80” -v
/c/Users/danielfe/Documents/DockerBook/ProductCatalog/Product
Catalog/src/ProductCatalog:/app thedanfernandez/productcatalog
```

c:\Users\<name>\Documents\DockerBook\catalog\ProductCatalog\src\ProductCatalog

/c/Users/<name>/Documents/DockerBook/catalog/ProductCatalog/src/ProductCatalog

```
docker run -i -t -p 80:80 -e “server.urls=http://*:80” --entrypoint /
bin/bash -v
/c/Users/danielfe/Documents/DockerBook/catalog/ProductCatalog/src/
ProductCatalog:/app thedanfernandez/productcatalog
```

```
c:\Users\<username>\Documents\DockerBook\catalog\src\ProductCatalog\
docker-compose.yml
```

~/DockerBook/catalog/ProductCatalog/src/ProductCatalog/docker-compose.yml.

```
productcatalog:
  image: "thedanfernandez/productcatalog"
  ports:
    - "80:80"
  tty: true
  links:
    - elasticsearch
  environment:
    - server.urls=http://*:80
elasticsearch:
  image: "elasticsearch"
  ports:
    - "9200:9200"
```

```
Public Iconfiguration Configuration { get; private set; }
public Startup(IapplicationEnvironment env)
{
  var builder = new ConfigurationBuilder(env.ApplicationBasePath)
  .AddJsonFile("config.json")
  .AddEnvironmentVariables();
  Configuration = builder.Build();
}
```

```
docker run -t -i -p 80:80 --entrypoint=/bin/bash thedanfernandez/
productcatalog
```

```
docker exec bbd8 cat docker-entrypoint.sh
```

```
{
   "$schema": "http://schema.management.azure.com/schemas/2015-01-01/
deploymentTemplate.json#",
   "contentVersion": "",
   "parameters": {  },
   "variables": {  },
   "resources": [  ],
   "outputs": {  }
}
```

```
"variables": {
  "location": "[resourceGroup().location]",
"masterDNSPrefix":"[concat(parameters('dnsNamePrefix'),'mgmt')]",
"agentCount": "[parameters('agentCount')]"
}
```

```
"parameters": {
  "agentCount": {
  "type": "int",
  "defaultValue": 1,
  "metadata": {
    "description": "The number of Mesos agents for the cluster."
  },
  "minValue":1,
  "maxValue":40
  },
"masterCount": {
  "type": "int",
  "defaultValue": 1,
  "allowedValues": [1, 3, 5],
  "metadata": {
    "description": "The number of Mesos masters for the cluster."
  }
}
```

```
"variables": {
  "masterDNSPrefix":"[concat(parameters('dnsNamePrefix'),'mgmt')]",
  "agentDNSNamePrefix":"[concat(parameters('dnsNamePrefix'),'agents')]"
  "agentCount": "[parameters('agentCount')]",
  "masterCount": "[parameters('masterCount')]",
  "agentVMSize": "[parameters('agentVMSize')]",
  "sshRSAPublicKey": "[parameters('sshRSAPublicKey')]",
  "adminUsername": "azureuser"
}
```

```
"resources": [
  {
    "apiVersion": "2015-11-01-preview",
    "type": "Microsoft.ContainerService/containerServices",
    "location": "[resourceGroup().location]",
    "name":"[concat('containerservice-',resourceGroup().name)]",
    "properties": {
      "orchestratorProfile": {
        "orchestratorType": "Swarm"
      }
    }
  }
]
```

```
"outputs": {
  "masterFQDN": {
    "type": "string",
    "value":
"[reference(concat('Microsoft.ContainerService/containerServices/',
'containerservice-', resourceGroup().name)).masterProfile.fqdn]"
  },
  "agentFQDN": {
    "type": "string",
    "value": "[reference(concat('Microsoft.ContainerService/
containerServices/', 'containerservice-', resourceGroup().name)).
agentPoolProfiles[0].fqdn]"
  }
}
```

```
{
  "apiVersion": "2015-11-01-preview",
  "type": "Microsoft.ContainerService/containerServices",
  "location": "[resourceGroup().location]",
  "name":"[concat('containerservice-',resourceGroup().name)]",
  "properties": {
    "orchestratorProfile": {
      "orchestratorType": "Mesos"
    },
    "masterProfile": {
      "count": "[variables('masterCount')]",
      "dnsPrefix": "[variables('mastersEndpointDNSNamePrefix')]"
    },
    "agentPoolProfiles": [
      {
        "name": "agentpools",
        "count": "[variables('agentCount')]",
        "vmSize": "[variables('agentVMSize')]",
        "dnsPrefix": "[variables('agentsEndpointDNSNamePrefix')]"
       }
     ],
     "linuxProfile": {
      "adminUsername": "[variables('adminUsername')]",
      "ssh": {
        "publicKeys": [
          {
            "keyData": "[variables('sshRSAPublicKey')]"
          }
        ]
      }
    }
  }
}
```

```
C:\> New-AzureResourceGroupDeployment -Name ExampleDeployment
-ResourceGroupName ExampleResourceGroup -TemplateFile
<PathOrLinkToTemplate> -TemplateParameterFile
<PathOrLinkToParameterFile>
```

```
azure group deployment create -f <PathToTemplate> -e
<PathToParameterFile> -g ExampleResourceGroup -n ExampleDeployment
```

```
azure group deployment create --template-uri <repository item URI>
```

```
azure group deployment create --template-uri
https://raw.githubusercontent.com/Azure/azure-quickstart-templates/
master/101-create-storage-account-standard/azuredeploy.json
```

```
$ docker -H tcp://<manager DNS name>-manage.<location>.cloudapp
.azure.com:2375
```

```
...
"parameters": {
    "newStorageAccountName": {
      "value": "uniqueStorageAccount"
    }
    "location": {
      "value": "West US"
    },
    "adminUsername": {
      "value": "username"
    },
    "adminPassword": {
      "value": "password"
    },
    "dnsNameForPublicIP": {
      "value": "uniqueDNS"
    }
}
```

```
"properties": {
 "hardwareProfile": {
    "vmSize": "[parameters('vmSize')]"
},
...
```

```
    {
      "apiVersion": "2015-05-01-preview",
      "type": "Microsoft.Compute/virtualMachines",
      "name": "[variables('vmName')]",
      "location": "[parameters('location')]",
      "tags": {
          "environment": "[parameters('environment)]"
          "location": "[parameters(location)]"
          "dept": "[parameters(dept)]"
      }
...
```

Docker daemon --label io.flak.storage="ssd"

```
Docker run -d \
--label io.flak.environment="dev" \
--label io.flak.dept="finance" \
--label io.flak.location="westus" \
nginx
```

```
Docker ps --filter "label=io.flak.environment=dev"
```

```
./docker-monitor -t 500 -e den@contoso.com
```

[ERROR] [2345] another message about the event - cid=1

{type:"info",thread:"2345",activityId:"1",message:"my message"}

```
{
    "storageAccountName":"the name of the Azure storage account
where the data is being persisted",
    "storageAccountKey":"the key of the account"
}
```

```
azure vm extension set vm_name LinuxDiagnostic Microsoft.
OSTCExtensions 2.* --private-config-path PrivateConfig.json
```

```
  "dependencies": {
    "Microsoft.AspNet.Mvc": "6.0.0-rc1-final",
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final",
    "Microsoft.Extensions.Configuration.FileProviderExtensions":
"1.0.0-rc1-final",
    "Microsoft.Extensions.Configuration.Json": "1.0.0-rc1-final",
    "Microsoft.Extensions.Logging": "1.0.0-rc1-final",
    "Microsoft.Extensions.Logging.Console": "1.0.0-rc1-final"
...
```

```
public Startup()
{
    Configuration = new Configuration()
    .AddJsonFile("config.json")
    .AddEnvironmentVariables();
}           ...
```

```
  {
    "AppSettings": {
      "SiteTitle": "My Website"
    },
    "ConnectionStrings": {
      "SqlDbConnection":
"Server=(localdb)\\mssqllocaldb;Database=Products;Trusted_
Connection=True;MultipleActiveResultSets=true"
    }

  }
```

```
var conn =
Configuration.Get<string>(["ConnectionStrings:SqlDbConnection");
```

```
  "scripts": {
    "prepublish": [ "npm install", "bower install", "gulp clean",
"gulp min" ]
  }
```

```
gulp.task("min", ["min:js", "min:css"]);
```

```
gulp.task("min:js", function () {
    gulp.src([paths.js, "!" + paths.minJs], { base: "." })
        .pipe(concat(paths.concatJsDest))
        .pipe(uglify())
        .pipe(gulp.dest("."));
});

gulp.task("min:css", function () {
    gulp.src([paths.css, "!" + paths.minCss])
        .pipe(concat(paths.concatCssDest))
        .pipe(cssmin())
        .pipe(gulp.dest("."));
});
```

```
 [HttpGet]
 public IEnumerable<Product> Get()
 {
     return DbContext.Products;
}
 [HttpGet]
 public async Task<IEnumerable<Product>> Get()
 {
     return await DbContext.Products.ToListAsync<Product>();
 }
```

```
[HttpGet]
public async Task<IEnumerable<Product>> Get()
{
    var result = await _context.Products
        .Take(25).ToArrayAsync();

    return result;
}
```

```
[HttpGet]
public async Task<IActionResult> Get()
{
    var result = await _context.Products
        .Take(25).ToArrayAsync();

    return new ObjectResult(result);
}
```

```
public class ProductsController : Controller
{
    [HttpGet("{id}")]
    public async Task<IActionResult> Get(int id)
    {
 var product =
  await _context.Products.FirstOrDefaultAsync(p =>
     p.ProductId == id);

        if (product == null)
        {
            return HttpNotFound();
        }

        return new JsonResult(product);
    }
```

```
        [HttpGet]
        public async Task<IEnumerable<Product>> Get()
        {
            return await
_context.Products.Take(25).ToListAsync<Product>();
        }
```