Microsoft

Programming Microsoft Azure Service Fabric

Second Edition

Professional

Haishi Bai

Microsoft

Programming Microsoft Azure Service Fabric
Second Edition

Haishi Bai

PROGRAMMING MICROSOFT AZURE SERVICE FABRIC (Second edition)
Published with the authorization of Microsoft Corporation by:
Pearson Education, Inc.

ISBN-13: 978-1-5093-0709-8
ISBN-10: 1-5093-0709-5

Library of Congress Control Number: 2018939252

1 18

Editor-in-Chief: Greg Wiegand

Executive Editor: Laura Norman

Development Editor: Kate Shoup/ Polymath Publishing

Managing Editor: Sandra Schroeder

Senior Project Editor: Tracey Croom/ Danielle Foster

Copy Editor: Dan Foster

Indexer: Valerie Haynes Perry

Proofreader: Scout Festa

Editorial Assistant: Cindy Teeters

Cover Designer: Twist Creative, Seattle

Compositor: Danielle Foster

Graphics: Vived Graphics

To the entire Microsoft Service Fabric team, who made such a great product.

—Haishi Bai

Contents at a Glance

Contents

About the Author

HAISHI BAI, principal software engineer at Microsoft, focuses on the Microsoft Azure compute platform, including IaaS, PaaS, networking, and scalable computing services.

Ever since he wrote his first program on an Apple II when he was 12, Haishi has been a passionate programmer. He later became a professional software engineer and architect. During his 21 years of professional life, he's faced various technical challenges and a broad range of project types that have given him rich experiences in designing innovative solutions to solve difficult problems.

Haishi is the author of a few cloud computing books, and he's an active contributor to a few open-source projects. He also runs a technical blog (http://blog.haishibai.com) with millions of viewers. His twitter handle is @HaishiBai2010.

Foreword

Service Fabric can be traced back to 2001, when I was trying to solve large-scale distributed system challenges such as leader election, quorum-based replication, and perfect failure detection. As I worked on projects in 2007, such as CloudDB (which morphed into Azure DB and is powered by Service Fabric), it became clear that a generic platform would be a valuable asset for empowering developers and enterprises to implement scalable and highly available distributed systems. In 2009, I began creating such a system to support large first-party and third-party workloads. This effort led to Service Fabric, which has proven itself in production for more than a decade. Service Fabric powers critical Microsoft services such as Azure DB, Cosmos DB, Skype for Business, Microsoft Intune, Azure Resource Providers (Compute RP, Storage RP, and Network RP), Azure Software Load Balancer, Azure Network Manager, Event Hubs, Event Grid, IoT Hubs, Azure Incident Manager, Azure Monitor, Bing Cortana, and more. As of March 2018, Service Fabric runs on about 4 million cores and individually monitors and fully lifecycle-manages about 10 million microservices. Technologies like Service Fabric are treated as trade secrets by many companies and are not made available to external customers. In March 2015, Service Fabric was released to the public. By making Service Fabric publicly available (and open-sourced in March 2018), Microsoft is living by the "first party == third party" principle.

Service Fabric is a comprehensive platform for building Internet-scale, high-throughput, low-latency services. Besides container orchestration, it solves many fundamental distributed systems problems, such as failure detection, replicated state machines, reliable message delivery, and so on. It allows developers to naturally decompose the business application into a logical set of microservices that are individually responsible for a single business function and interact with one another over well-defined protocols for implementing business workflows. Service Fabric lets developers focus on business logic and its associated state by abstracting away machine and distributed systems details with many built-in transactionally consistent reliable data structures like dictionaries and queues that survive process crashes and machine failures. It enables programmers to think and program exactly like they do today by replacing locks with transactions. Programmers assume that the process hosting their code never crashes, and the data structures storing their state never lose or corrupt their data. With its ability to run on any OS and on any cloud, including on-premises and Edge, developers preserve their code investments across a wide variety of deployment targets. Put differently, Service Fabric allows programming large-scale applications to be just like writing simple applications.

I have known Haishi for about three years now. He has keenly followed the evolution of Service Fabric as a public service and has a deep understanding of its developer and operational aspects. He also understands intuitively how various Service Fabric layers and subsystems combine to provide the solutions to many distributed-systems problems. His first edition of this book focused on Service Fabric programming models and design patterns. This second edition is a more comprehensive follow-up companion book on Service Fabric that focuses more on the developer and operational aspects of Service Fabric and newer parts of the Service Fabric-like containers, Linux support, and more.

If you are interested in microservices and stateless and stateful variants and are using or intend to use Service Fabric for developing microservices, this a must-read book for you.

—Gopal Kakivaya
CVP, Microsoft Azure Development

Introduction

Azure Service Fabric is Microsoft's platform as a service (PaaS) offering for developers to build and host available and scalable distributed systems. Microsoft has used Service Fabric for years to support some of Microsoft's cloud-scale applications and Azure services such as Skype for Business, Cortana, Microsoft Intune, Azure SQL Database, and Azure Cosmos DB. The same platform is now available as an open-source project for you to write your own highly available and highly scalable services.

Programming Microsoft Azure Service Fabric is designed to get you started and quickly productive with Azure Service Fabric. This book covers fundamentals, practical architectures, and design patterns for various scenarios, such as intelligent cloud, intelligent edge, big data, and distributed computing. For the fundamentals, this book provides detailed step-by-step walkthroughs that guide you through typical DevOps tasks. For design patterns, this book focuses on explaining the design philosophy and best practices with companion samples to get you started and moving in the right direction.

Instead of teaching you how to use Azure Service Fabric in isolation, the book encourages developers to make smart architecture choices by incorporating existing Azure services. When appropriate, this book briefly covers other Azure services relevant to particular scenarios.

Who should read this book

This book is intended to help new or experienced Azure developers get started with Azure Service Fabric. This book is also useful for architects and technical leads using Azure Service Fabric and related Azure services in their application architecture.

Service Fabric is under continuous development, and its momentum will only accelerate by community contributions. The second edition of this book offers the latest development of Service Fabric at the time of this writing. For the latest updates, consult the book's companion resource repository (*https://github.com/Haishi2016/ProgrammingServiceFabric*) and Service Fabric's online documentation (*https://docs.microsoft.com/azure/service-fabric*). Although the precise operational steps and programming APIs might change, the design patterns presented in this book should remain relevant into the foreseeable future.

Assumptions

This book expects that you are proficient in .NET, especially C# development. This book covers a broad range of topics and scenarios, especially in later chapters. Prior understanding of DevOps, application life cycle management (ALM), IoT, big data, and machine learning will help you get the most out of this book.

Although no prior Azure knowledge is required, experience with the Azure software development kit (SDK), Azure management portal, Azure PowerShell, Azure command line interface (CLI), and other Azure services definitely will be helpful.

This book might not be for you if…

This book might not be for you if you are a beginner in programming. This book assumes you have previous experience in C# development and ASP.NET development. Although this book covers topics in service operations, its primary audience is developers and architects, not IT professionals.

Organization of this book

This book is divided into five parts, each of which focuses on a different aspect of Azure Service Fabric. Part I, "Fundamentals," provides complete coverage of designing and developing Service Fabric applications using stateless services, stateful services, and reliable actors. Part II, "Service Life Cycle Management," focuses on the operations side and introduces how to manage Service Fabric clusters and how to manage, test, and diagnose Service Fabric applications. Part III, "Linux and Containers," introduces Service Fabric Linux programming with Java and support for Docker containers. Part IV, "Workloads and Design Patterns," introduces patterns and scenarios including practical design patterns and best practices in implementing typical application scenarios including scalable web applications, IoT, big data, and multi-tenant applications. Finally, Part V, "Advanced Topics," covers three advanced topics: serverless computing, machine learning, and the intelligent cloud and the intelligent edge.

Finding your best starting point in this book

This book is an introduction to Service Fabric. It is recommended that you read the chapters in the first two parts sequentially. Then, you can pick the topics that interest you in Parts III, IV, and V.

If you are	Follow these steps
New to Service Fabric	Read through Part I and Part II in order.
Interested in applying Service Fabric in IoT scenarios	Focus on Chapter 20.
Interested in building scalable web applications	Focus on Chapters 7, 15, and 16. You may also want to read Chapter 17 for integrations with other Azure services, and Chapter 18 for serverless options.
Interested in machine learning	Focus on Chapter 19.
Interested in operating a Service Fabric cluster	Chapters 8, 9, 10, and 11 introduce related tools and services. You may also want to browse through Chapters 5, 6, and 7 to understand Service Fabric application characteristics.
Interested in the actor programming model	Focus on Chapter 4. Also browse through chapters in Part III because these chapters cover several actor-based design patterns.
Interested in Service Fabric container integration	Focus on Chapters 13 and 14.
Interested in Service Fabric Linux development	Focus on Chapter 12.

Some of this book's chapters include hands-on samples that let you try out the concepts just learned. Regardless of which sections you choose to focus on, be sure to download and install the sample applications on your system (see the "Downloads" section on the next page).

System requirements

You will need the following hardware and software to run the sample code in this book:

- Windows 8/Windows 8.1, Windows Server 2012 R2, or Windows 10.
- Visual Studio 2015 or Visual Studio 2017.
- The latest Service Fabric SDK for Visual Studio 2015 or Visual Studio 2017 (install via Web PI).
- The latest version of Azure SDK (2.8 or above, install via Web PI).
- The latest version of Azure PowerShell (1.0 or above, install via Web PI).
- The latest version of Azure CLI.
- 4 GB (64-bit) RAM.

- 50 GB of available hard disk space.
- An active Microsoft Azure subscription. You can get a free trial from www.azure.com.
- An Internet connection to use Azure and to download software or chapter examples.
- An Ubuntu 14.0 or above machine or virtual machine for Linux-based exercises.

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 2015, Visual Studio 2017, and related SDKs and tools.

Downloads: Code samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All sample projects can be downloaded from the following page:

https://aka.ms/AzureServFabric2e/downloads

You can also find the latest sample projects from the book's companion source repository:

https://github.com/Haishi2016/ProgrammingServiceFabric

Using the code samples

This book's webpage contains all samples in this book, organized in corresponding chapter folders. It also contains a V1-Samples folder that contains samples from the original version of the book, thanks to Alessandro Avila's contributions.

- **Chapter 1** This folder contains samples from Chapter 1.

  HelloWorldApplication: The "Hello, World!" application.

- **Chapter 2** This folder contains samples from Chapter 2.

  CalculatorApplication: The calculator application used in the communication stack samples.

  gRPCApplication: The calculator application using gRPC.

  StatelessApplication: The ASP.NET Core Web API stateless application.

- **Chapter 3** This folder contains samples from Chapter 3.

  SimpleStoreApplication: The simple store application.

- **Chapter 4** This folder contains samples from Chapter 4.

  ActorTicTacToeApplication: The tic-tac-toe game using actors.

  CarApplication: The simple car simulation program using actors.

- **Chapter 5** This folder contains samples from Chapter 5.

  ConsoleRedirectTestApplication: The sample application used in package format samples.

  HelloWorldWithData: The sample application with a data package.

  NodeJsHelloWorldApplication: The sample application hosts a Node.js application.

  ResourceGovernance: The sample application with resource governance policy.

  UpgradeProcess: The rolling update sample application.

- **Chapter 6** This folder contains samples from Chapter 6.

  BadApplication: The sample unreliable application using in-memory states.

  ChaosTest: The sample application that drives a chaos test.

  ConfigurationUpdate: The sample application that responds to configuration updates.

  Diagnostics: The sample application used in the Azure Diagnostics sample.

  FailoverTest: The sample application used in the failover sample.

- **Chapter 7** This folder contains samples from Chapter 7.

  CustomSerializerTest: The sample application that uses a custom serializer.

- **Chapter 8** This folder contains samples from Chapter 8.

  DeploymentTest: The application used in the deployment sample.

- **Chapter 10** This folder contains samples from Chapter 10.

  ApplicationInsightsTestApplication: The sample application that uses Application Insights.

CustomHealthReportApplication: The sample application that sends custom health reports.

EventFlowTestApplication: The sample application that uses Event Flow for diagnostics.

OMSTestApplication: The application used in the OMS sample.

- **Chapter 11** This folder contains samples from Chapter 11.

  SudokuApplication: The sample Sudoku application.

- **Chapter 12** This folder contains samples from Chapter 12.

  ActorApplication: The sample actor application in Java.

  CalculatorApplication: The sample calculator application in Java.

  GuestPythonApplication: The sample Python application hosted as a guest executable.

  HelloWorldLinux: The "Hello, World!" application in Java.

  StatefulApplication: The sample stateful application in Java.

- **Chapter 13** This folder contains samples from Chapter 13.

  Docker-HelloWorld-Windows: The sample ASP.NET Core web application in Windows container.

  DockerCompose-HelloWorld: The sample container application described by a Docker Compose file.

  Minecraft: The HA Minecraft server deployment using persistent volume.

- **Chapter 14** This folder contains samples from Chapter 14.

  IrisApp: The sample two-tiered application that uses the Python Sklearn library in containers.

  ServiceMesh: The service mesh sample that uses Envoy.

  Watchdog: The sample service that uses a C# watchdog to monitor a containerized sample Java application.

- **Chapter 15** This folder contains samples from Chapter 15.

  PortSharing: The port sharing sample application.

  TenantManager: The sample implementation of the tenant manager pattern.

- **Chapter 16** This folder contains samples from Chapter 16.

  ECommerceApp: The sample e-commerce application.

  NumberConverterApp: The number converter service.

- **Chapter 17** This folder contains samples from Chapter 17.

  AudioTranscriptionApp: The audio transcription application that uses Bing Speech API.

- **Chapter 18** This folder contains samples from Chapter 18.

  aci: The Azure Container Instance example.

  ReactiveActors: The sample reactive application using actors.

- **Chapter 19** This folder contains samples from Chapter 19.

  ArchiBot: The architecture bot sample that uses Bot Framework and Cosmos DB.

To complete an exercise, access the appropriate chapter folder in the root folder and open the project file. If your system is configured to display file extensions, C# project files use the .csproj file extension.

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

https://aka.ms/AzureServFabric2e/errata

If you discover an error that is not already listed, please submit it to us at the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *https://support.microsoft.com*.

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

PART III

Linux and Containers

Service Fabric provides a microservices programming model that you can use to build native cloud microservices applications. However, as discussed in Part I, implementing microservices doesn't mandate a specific programming model. If an application can be packaged in a self-contained format and be consistently deployed on different environments, it can enjoy many cloud benefits, such as failover, scaling, and load balancing. While packaging application artifacts isn't hard, making sure the application has all its dependencies is not easy. A legacy application may have dependencies on external libraries and services; it may assume specific folder structures; it may require specific environment variables; and for Windows, it may depend on certain registry values. It's impossible to isolate and package such system-level dependencies by a simple file-based package mechanism.

One way to package these system-level dependencies is to use desired state configuration (DSC). With DSC, such dependencies are captured as solution-specific metadata. When an application is deployed, the metadata is checked against the actual host environment. If any discrepancies are found, predefined scripts are executed to bring the host environment into the desired state. For example, if an application requires a specific DirectX version, the requirement can be captured as

DSC metadata. And when the application is deployed on a new host, the DSC system will check whether DirectX is installed. If not, it will run predefined scripts (most DSC systems have "ingredients" that perform common configuration tasks such as installing software packages) to make sure the specific version of DirectX is installed. Still, bringing an arbitrary machine state to a desired state is a hard job. In some cases, two applications are simply incompatible and cannot be installed on the same host. For example, if application A and application B use a c:\data folder, they can't co-exist on the same host without interfering with each other.

Virtualization is an effective way of addressing these challenges. With virtualization, an application resides on a virtualized operating system that virtualizes process spaces, file systems, and registries (for Windows). In such an environment, because the application has exclusive access to the entire virtualized environment, it doesn't need to worry about any potential conflicts with any other applications. The use of virtual machines (VMs) is a mature virtualization technique that has been broadly used in both on-premises and cloud datacenters. However, VMs are quite heavy. They require significant resources, and it takes a long time to provision, update, and deprovision a VM. Hence, VMs don't provide the application mobility microservices requires.

Containers provide fine-grained isolation by isolating processes, files, and registries. Containers running on the same host share the same system kernel and can be launched and destroyed in the same way as regular processes. For Linux systems, this means sub-second launch times (Windows containers take longer to launch, but it's still much faster than booting up a VM.) You can also pack many containers on the same host to gain high compute density. Containers are perfect for microservices because they package applications into lightweight, isolated, and consistently deployable units. This is the exact application mobility microservices requires to perform failovers, replications, scaling, and load balancing.

Given how powerful containers are, it makes sense for Service Fabric to provide native support for them. Furthermore, as a microservices platform, Service Fabric needs to embrace microservices that are not written using the Service Fabric programming model. In the past two years, Service Fabric has built up first-class support for Linux and containers. This will be the focus of this part of the book.

CHAPTER 12

Service Fabric on Linux

Service Fabric running on Linux may come as a surprise. Why would Microsoft invest in non-Windows platforms? The reality, however, is that Service Fabric on Linux resonates with the openness of Azure strategy. Azure has never been just for Microsoft technologies. It's an inclusive platform that welcomes all types of workloads on all technical stacks. In fact, a huge portion of Azure compute power resides on Linux. The percentage of Linux VMs on Azure was 33% in 2016 and 40% in 2017, and it continues to increase.

As a microservices platform, Service Fabric must embrace not only Windows-based workloads, but also Linux-based workloads. In the past two years, the Service Fabric team has built up native Linux support with Java tooling. This chapter introduces the Linux development experiences using the Service Fabric programming model. Chapter 13 and Chapter 14 focus on containers and container orchestrations.

> **Note** At the time of this writing, Service Fabric on Linux has just recently become generally available. Tooling experiences and product behaviors are subject to change. Please visit this book's companion GitHub repository for updated samples.

Service Fabric Hello, World! on Linux

The Service Fabric team has chosen Java as its primary programming language on Linux. In this section, you'll learn how to set up a development environment on Linux and use Java to write a simple Service Fabric application.

Setting Up Your Linux Development Environment

Follow these steps to set up your Linux development environment. (These instructions are based on Ubuntu 16.04.)

1. To install the Service Fabric runtime, Service Fabric common SDK, and a sfctl CLI, use the following script:

```
sudo curl -s https://raw.githubusercontent.com/Azure/service-fabric-scripts-and-
templates/master/scripts/SetupServiceFabric/SetupServiceFabric.sh | sudo bash
```

2. Use this script to set up a local cluster:

```
sudo /opt/microsoft/sdk/servicefabric/common/clustersetup/devclustersetup.sh
```

3. After the cluster has been configured, open a web browser and navigate to http://localhost:19080/Explorer. The Service Fabric Explorer should open. Or, if you want to try out sfctl, you can use the following command to obtain a list of node names:

```
sfctl cluster select –endpoint http://localhost:19080
sfct node list | grep name
```

4. Service Fabric uses Yeoman to scaffold Service Fabric applications. Yeoman is an open-source tool originally designed for this task. Use the following commands to install and configure Yeoman with Service Fabric application generators:

```
sudo apt-get install npm
sudo apt install nodejs-legacy
sudo npm install -g yo
sudo npm install -g generator-azuresfcontainer
sudo npm install -g generator-azuresfguest
```

5. If you plan to build Service Fabric services using Java, use the following commands to install JDK 1.8 and Gradle:

```
sudo apt-get install openjdk-8-jdk-headless
sudo apt-get install gradle
```

6. For the IDE, Service Fabric chose Eclipse for Java development. To install Eclipse, download the package from www.eclipse.org/downloads/eclipse-packages (this book uses the Oxygen.1 version), extract all files from the package, and launch **eclipse-inst**.

7. Service Fabric provides an Eclipse plug-in to facilitate application creation. After you install Eclipse, launch it, open the **Help** menu, and choose **Install New Software**.

8. In the **Work With** box, type **http://dl.microsoft.com/eclipse**. Then click the **Add** button.

9. Select the **ServiceFabric plug-in**. Then follow the wizard to install the plug-in.

Note If you want to enable the desktop UI on your Azure Ubuntu VM, use the following commands (tested on Ubuntu 16.04):

```
sudo apt-get install xrdp
sudo apt-get update
sudo apt-get install xfce4
echo xfce4-session >~/.xsession
sudo service xrdp restart
```

Then configure networking on the VM to allow inbound RDP connections through port 3389. After that, you should be able to connect to your Ubuntu desktop using RDP.

Hello, World! Again

Now it's time to send greetings to a brand-new world. In the following exercise, you'll create a new Service Fabric stateless service using Eclipse. Follow these steps:

1. In Eclipse, open the **File** menu, choose **New**, and select **Other**.
2. Expand the **Service Fabric** folder, select **Service Fabric Project**, and click **Next**. (See Figure 12-1.)

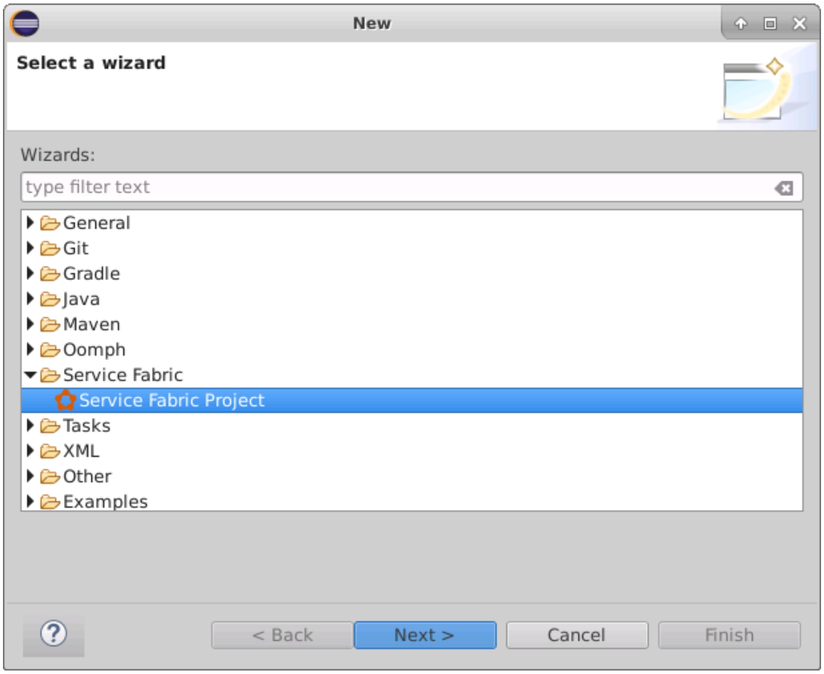

FIGURE 12-1 The Eclipse New Project wizard.

3. In the next screen, type **HelloWorldLinux** in the **Project Name** box and click **Next**.
4. In the Add Service screen, select the **Stateless Service** template, type **HelloWorldService** in the **Service Name** box, and click **Finish** to add the service. (See Figure 12-2.)

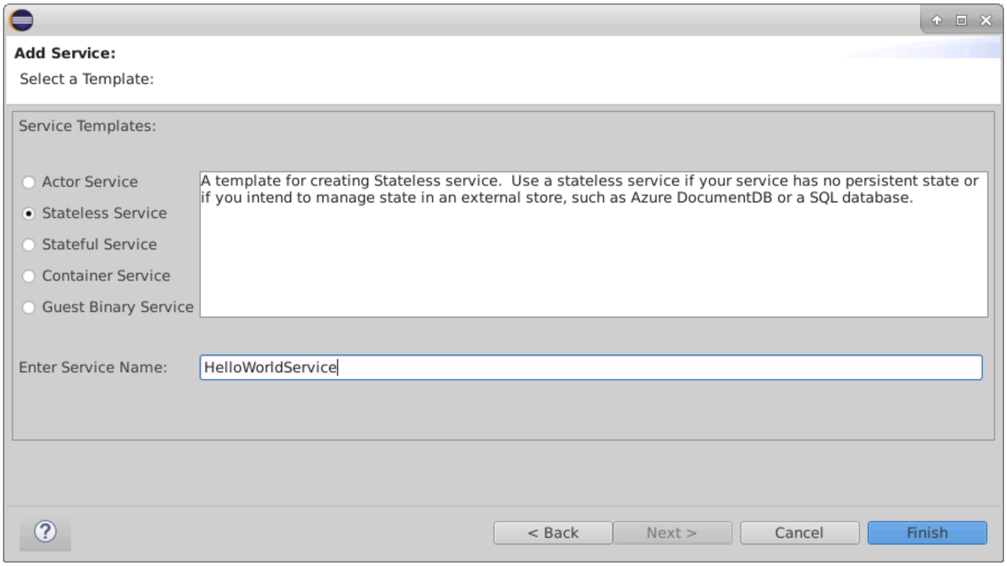

FIGURE 12-2 Add a service.

5. When the wizard prompts you to open the Service Fabric perspective, click **Open Perspective**.
6. After the application is created, poke around the package tree to familiarize yourself with the package structure.

   Fortunately, the Java project is similar to a C# project. You can see how instance listeners are created, how the `runAsync` method is implemented in HelloWorldServiceService.java, and how the service is registered in HelloWorldServiceServiceHost.java. (See Figure 12-3.)

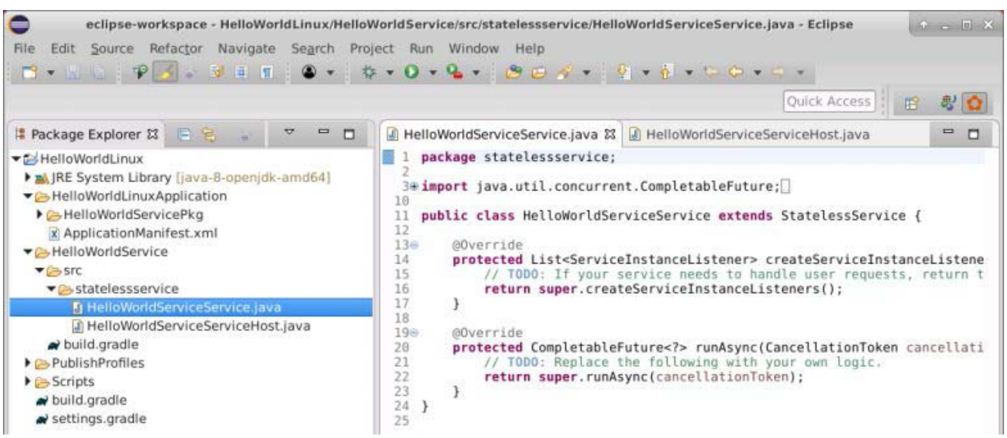

FIGURE 12-3 Java package structure.

7. In Package Explorer, right-click **HelloWorldLinux**, select **Service Fabric**, and choose **Deploy Application** to build and deploy the application. After the application is deployed, you should see your service instance running through the Service Fabric Explorer.

> **Note** Provisioning a Service Fabric cluster consumes about 22 GB of your system drive. If you are running low on disk space, you may want to move the cluster to a different volume. Here's how:
>
> ```
> cd /opt/microsoft/sdk/servicefabric/common/clustersetup
> sudo ./devclustercleanup.sh
> sudo rm -rf /home/sfuser/sfdevcluster
> sudo ./devclustersetup.sh --clusterdataroot=/some/other/volume
> ```

8. Open **HelloWorldService\src\statelessservice\HelloWorldServiceService.java** and examine the stateless service implementation. The scaffolded implementation isn't exciting; indeed, it doesn't do anything:

```
protected CompletableFuture<?> runAsync(CancellationToken cancellationToken) {
        return super.runAsync(cancellationToken);
}
```

9. In a moment, you'll modify the `runAsync` method to make it behave in the same way as the default C# stateless service and maintain an incrementing local counter. You'll also add a `FileHandler` to record log entries into files under the services' log folder. First, though, you'll need to import the following namespaces:

```
import java.util.logging.FileHandler;
import java.util.logging.SimpleFormatter;
```

10. Issue the following command to add a logger as a static member of the `HelloWorldServiceService` class:

```
private static final Logger logger =
    Logger.getLogger(HelloWorldServiceService.class.getName());
```

11. Modify the `runAsync` method as shown in the following code:

```
@Override
protected CompletableFuture<?> runAsync(CancellationToken cancellationToken) {
        try
        {
                String logPath = super.getServiceContext()
                    .getCodePackageActivationContext().getLogDirectory();
                FileHandler handler = new FileHandler(logPath
                    + "/mysrv-log.%u.%g.txt", 1024000, 10, true);
                handler.setFormatter(new SimpleFormatter());
                handler.setLevel(Level.ALL);
                logger.addHandler(handler);
        } catch (Exception exp) {
                logger.log(Level.SEVERE, null, exp);
        }
```

```
        return CompletableFuture.runAsync(() -> {
                try
                {
                        int iteration = 0;
                        while (!cancellationToken.isCancelled()) {
                                logger.log(Level.INFO, "Working-" + iteration++);
                                Thread.sleep(1000);
                        }
                } catch (Exception exp) {
                        logger.log(Level.SEVERE, null, exp);
                }
        });
}
```

Note a couple of things in the preceding code:

- You can use the `getServiceContext()` method to obtain the service context, through which you can navigate an object tree that is very similar to the .NET object tree to obtain contextual information—for example to obtain the service's log directory using the object model.
- The `runAsync` method is supposed to return a `java.util.concurrent.Completable` `Future`<T> instance.

12. Build and deploy the application to your local cluster.
13. Locate the log path of your service instance on the hosting node. You'll find generated log files under the services log folder.

Using Communication Listeners

To create a service that listens to client requests, you need to create and register a `CommunicationListener` implementation. This process is very similar to what you did in Chapter 2. Perform the following steps to create a Java-based calculator service that provides a REST API for add and `subtract` calculations:

1. In Eclipse, create a new Service Fabric application named CalculatorApplication with a stateless service named Calculator.
2. Add a new `CalculatorServer` class to the project. This class contains nothing specific to Service Fabric. It uses `com.sun.net.httpserver.HttpServer` to handle add and `subtract` requests from clients.

```
package statelessservice;

import com.sun.net.httpserver.*;
import java.net.InetSocketAddress;
import java.io.IOException;
import java.io.OutputStream;
import java.io.UnsupportedEncodingException;
import java.util.HashMap;
```

```
import java.util.Map;

public class CalculatorServer {
        private HttpServer server;
    private int port;
    public CalculatorServer(int port) {
            this.port = port;
    }
    public void start() throws IOException {
            server = HttpServer.create(new InetSocketAddress(port),0);
            HttpHandler add = new HttpHandler() {
                    @Override
                    public void handle(HttpExchange h) throws IOException {
                            byte[] buffer = CalculatorServer.handleCalculation
                                (h.getRequestURI().getQuery(), "add");
                            h.sendResponseHeaders(200, buffer.length);
                            OutputStream os = h.getResponseBody();
                            os.write(buffer);
                            os.close();
                    }
            };
            HttpHandler subtract = new HttpHandler() {
                    @Override
                    public void handle(HttpExchange h) throws IOException {
                            byte[] buffer = CalculatorServer.handleCalculation
                                (h.getRequestURI().getQuery(), "subtract");
                            h.sendResponseHeaders(200, buffer.length);
                            OutputStream os = h.getResponseBody();
                            os.write(buffer);
                            os.close();
                    }
            };
            server.createContext("/api/add", add);
            server.createContext("/api/subtract", subtract);
            server.setExecutor(null);
            server.start();
    }
    public void stop() {
            server.stop(10);
    }
    public static Map<String, String> queryToMap(String query) {
            Map<String, String> map = new HashMap<String, String>();
            for (String param: query.split("&")) {
                    String pair[] = param.split("=");
                    if (pair.length > 1) {
                            map.put(pair[0], pair[1]);
                    } else {
                            map.put(pair[0], "0");
                    }
            }
            return map;
    }
    public static byte[] handleCalculation(String query, String type)
        throws UnsupportedEncodingException {
            byte[] buffer = null;
            Map<String, String> parameters = CalculatorServer.queryToMap(query);
            int c = 0;
```

```
            try
            {
                int a = Integer.parseInt(parameters.get("a"));
                int b = Integer.parseInt(parameters.get("b"));
                if (type.equals("add")) {
                        c = a + b;
                } else {
                        c = a - b;
                }
                buffer = Integer.toString(c).getBytes("UTF-8");
            } catch (NumberFormatException e) {
                    buffer = ("Invalid parameters").getBytes("UTF-8");
            }
            return buffer;
    }
}
```

3. Add a `WebCommunicationListener` class to the project. This class implements `microsoft.servicefabric.services.communication.runtime.CommunicationListener` and overrides the openAsync, `closeAsync`, and `abort` methods.

```
package statelessservice;

import java.util.concurrent.CompletableFuture;
import java.io.IOException;
import microsoft.servicefabric.services.communication.runtime.CommunicationListener;
import microsoft.servicefabric.services.runtime.StatelessServiceContext;
import system.fabric.description.EndpointResourceDescription;
import system.fabric.CancellationToken;

public class WebCommunicationListener implements CommunicationListener {
    private StatelessServiceContext context;
    private CalculatorServer server;
    private String webEndpointName = "ServiceEndpoint";
    private int port;
    public WebCommunicationListener(StatelessServiceContext context) {
            this.context = context;
            EndpointResourceDescription endpoint =
                this.context.getCodePackageActivationContext().getEndpoint
                    (webEndpointName);
        this.port = endpoint.getPort();
    }

    @Override
    public CompletableFuture<String> openAsync(CancellationToken cancellationToken) {
            CompletableFuture<String> str = new CompletableFuture<>();
            String address = String.format("http://%s:%d/api",
                this.context.getNodeContext().getIpAddressOrFQDN(), this.port);
            str.complete(address);
            try
            {
                server = new CalculatorServer(port);
                server.start();
            } catch (IOException e) {
                    throw new RuntimeException(e);
            }
```

```
            return str;
    }

    @Override
    public CompletableFuture<?> closeAsync(CancellationToken cancellationToken) {
            CompletableFuture<Boolean> task = new CompletableFuture<>();
            task.complete(Boolean.TRUE);
            if (server != null) {
                    server.stop();
            }
            return task;
    }

    @Override
    public void abort() {
            if (server != null) {
                    server.stop();
            }
    }
}
```

4. Modify `CalculatorService` to return `WebCommunicationListener` from the overridden `createServiceInstanceListeners` method:

```
import java.util.ArrayList;
...
public class CalculatorService extends StatelessService {
    @Override
    protected List<ServiceInstanceListener> createServiceInstanceListeners() {
        ArrayList<ServiceInstanceListener> listeners = new ArrayList();
        listeners.add(new ServiceInstanceListener((context) -> {
                return new WebCommunicationListener(context);
        }));
        return listeners;
    }
}
```

5. Modify the CalculatorApplicationApplication\CalculatorPkg\ServiceManifest.xml file to define an endpoint resource named `ServiceEndpoint`:

```
<Resources>
    <Endpoints>
        <Endpoint Name="ServiceEndpoint" Protocol="http" Port="8182" />
    </Endpoints>
</Resources>
```

6. Build and deploy the application. Afterward, you should be able to use a browser and send requests such as `http://localhost:8182/api/add?a=100&b=200` and `http://localhost:8182/api/subtract?a=100&b=200`. You should also get corresponding outputs (300 and –100).

Other Service Types and Frameworks

Using Service Fabric Java SDK to implement other service types, including stateful services, actor services, and guest application services, is very similar to using the .NET SDK. Of course, because of language differences, the Java SDK is adapted to work more naturally for Java developers.

> **Note** One productivity goal of the Service Fabric team is to enable popular and proven programming paradigms on Service Fabric. You can expect to see increasingly more languages and frameworks receive native support through Service Fabric tooling.

The next few sections provide a quick glimpse into how Java SDK supports different service types. To try out different service types, simply create a Service Fabric application and add a service with the type. The scaffolded code gives you quick examples on basic usage of the corresponding service types. You'll see a great similarity between the Java code and the .NET code.

Stateful Services

Interacting with state managers in Java SDK is slightly different from the .NET SDK. The Java SDK uses a `CompletableFuture`<T> type, which can be taken as an approximation of the .NET `Task`<T> type. Or, if you are familiar with promises, you can take it as an implementation of a promise. When you create a new stateful service, the SDK scaffolds a default `runAsync` method implementation, as shown in the following snippet:

```
@Override
protected CompletableFuture<?> runAsync(CancellationToken cancellationToken) {
    Transaction tx = stateManager.createTransaction();
    return this.stateManager.<String, Long>getOrAddReliableHashMapAsync("myHashMap")
        .thenCompose((map) -> {
        return map.computeAsync(tx, "counter", (k, v) -> {
            if (v == null)
                return 1L;
            else
                return ++v;
        }, Duration.ofSeconds(4), cancellationToken).thenApply((l) -> {
            return tx.commitAsync().handle((r, x) -> {
                if (x != null) {
                    logger.log(Level.SEVERE, x.getMessage());
                }
                try {
                    tx.close();
                } catch (Exception e) {
                    logger.log(Level.SEVERE, e.getMessage());
                }
                return null;
            });
        });
    });
}
```

This code first creates a new transaction. Then, it tries to get or add a `microsoft.servicefabric.data.collections.ReliableHashMap<K,V>` instance, which is equivalent to `IReliableDictionary<K,V>` in the .NET SDK. Finally, it tries to create or update the `"counter"` entry in the map and commits the transaction.

Actor Services

Actor services with Java SDK work in the same way as .NET-based actor services. To construct and use an actor service in Java, you need the same set of artifacts as in the .NET SDK:

- **Actor interface** An actor interface is defined as a regular Java interface that inherits a default `Actor` interface—for example:

```
public interface MyActor extends Actor {
    @Readonly
    CompletableFuture<Integer> getCountAsync();
    CompletableFuture<?> setCountAsync(int count);
}
```

- **Actor implementation** An actor implementation inherits from a `microsoft.servicefabric.actors.FabricActor` base class and implements the actor interface. The following snippet shows that the actor implementation in Java corresponds almost line by line with the .NET implementation:

```
@ActorServiceAttribute(name = "MyActorActorService")
@StatePersistenceAttribute(statePersistence = StatePersistence.Persisted)
public class MyActorImpl extends FabricActor implements MyActor {
    private Logger logger = Logger.getLogger(this.getClass().getName());
    public MyActorImpl(FabricActorService actorService, ActorId actorId){
        super(actorService, actorId);
    }
    @Override
    protected CompletableFuture<?> onActivateAsync() {
        logger.log(Level.INFO, "onActivateAsync");
        return this.stateManager().tryAddStateAsync("count", 0);
    }
    @Override
    public CompletableFuture<Integer> getCountAsync() {
        logger.log(Level.INFO, "Getting current count value");
        return this.stateManager().getStateAsync("count");
    }
    @Override
    public CompletableFuture<?> setCountAsync(int count) {
        logger.log(Level.INFO, "Setting current count value {0}", count);
        return this.stateManager().addOrUpdateStateAsync("count", count,
            (key, value) -> count > value ? count : value);
    }
}
```

- **Actor proxy for a client to connect to an actor service** The Java SDK provides a `microsoft.servicefabric.actors.ActorProxyBase` class, through which you can create actor proxies for your actor interface:

```
MyActor actorProxy = ActorProxyBase.create(MyActor.class, new ActorId("From Actor 1"),
    new URI("fabric:/ActorApplicationApplication/MyActorActorService"));
int count = actorProxy.getCountAsync().get();
System.out.println("From Actor:" + ActorExtensions.getActorId(actorProxy)
    + " CurrentValue:" + count);
actorProxy.setCountAsync(count+1);
```

Guest Binary Services

As on the Windows platform, you can package a guest binary and host it on Service Fabric as a stateless service. The following steps show you how to package a Python-based application as a guest binary service. In this simple example, you'll create a web server using Flask and then host the Python application as a stateless service on your local Service Fabric cluster.

1. Create a new Service Fabric application named GuestPythonApplication with a guest binary service named FlaskWebServer.

2. When asked for guest binary details, simply click the **Finish** button. You'll add these files manually later.

3. Use the following code, which uses Flask to implement a very simple web server, to create a new flaskserver.py file in the GuestPythonApplicationApplication\FlaskWebServerPkg\Code folder:

```
from flask import Flask
app = Flask("myweb")
@app.route("/")
def hello():
  return "Hello from Flask!"
app.run(host='0.0.0.0', port=8183, debug = False)
```

4. In the same folder, add a new launch.sh file to launch the web server. The following script first installs the flask module using pip. Then, it locates the path of the current script and feeds the correct server file path to Python. Strictly speaking, installing Flask should have been done in a setup entry point because it's a host environment configuration step. I'll leave this exercise to interested readers.

```
#!/bin/bash
sudo python -m pip install flask >> ../log/flask-install.txt 2>&1
pushd $(dirname "${0}") > /dev/null
BASEDIR=$(pwd -L)
popd > /dev/null
logger ${BASEDIR}
python ${BASEDIR}/flaskserver.py
```

5. Update the ServiceManifest.xml file as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<ServiceManifest Name="FlaskWebServerPkg" Version="1.0.0" xmlns="http://schemas.
microsoft.com/2011/01/fabric" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Description>Service that implements a FlaskWebServer service</Description>
  <ServiceTypes>
    <StatelessServiceType ServiceTypeName="FlaskWebServerType" UseImplicitHost="true"/>
  </ServiceTypes>
  <CodePackage Name="Code" Version="1.0.0">
    <EntryPoint>
      <ExeHost>
        <Program>launch.sh</Program>
        <Arguments />
        <WorkingFolder>CodePackage</WorkingFolder>
      </ExeHost>
    </EntryPoint>
    <EnvironmentVariables></EnvironmentVariables>
  </CodePackage>
  <ConfigPackage Name="Config" Version="1.0.0" />
  <DataPackage Name="Data" Version="1.0.0" />
  <Resources>
      <Endpoints>
          <Endpoint Name="ServiceEndpoint" Protocol="http" Port="8183" Type="Input"/>
      </Endpoints>
  </Resources>
 </ServiceManifest>
```

6. Build and deploy the application.

7. Using a web browser, navigate to http://localhost:8183/. You should see a "Hello from Flask!" message.

Using Yeoman

In addition to the Eclipse experience, Service Fabric provides a few generators that enable you to create Service Fabric applications using Yeoman. Yeoman (http://yeoman.io/) is an application scaffolding tool with an extensible generator ecosystem that hosts generators for various application types, including Service Fabric applications.

When you install the Service Fabric SDK, Yeoman is installed and configured automatically. To launch Yeoman, issue the yo command in a terminal.

To recreate the previous guest binary application in Yeoman, follow these steps:

1. Create a new ~/pythonflask folder and copy the flaskserver.py and launch.sh files into it.

2. Use the `yo azuresfguest` command to launch Yeoman with the azuresfguest generator. You should see output like that shown in Figure 12-4.

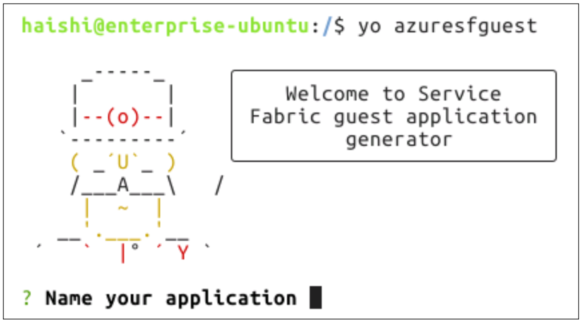

FIGURE 12-4 Use the azuresfguest generator with Yeoman.

3. Yeoman prompts you to provide information. Respond to these prompts as shown here:

   - **Name Your Application** GuestApp
   - **Name of the Application Service** Flask
   - **Source Folder of Guest Binary Artifacts** /home/*<your user name>*/pythonflask (Based on my tests, you need to provide the absolute path here.)
   - **Relative Path to Guest Binary in Source Folder** launch.sh
   - **Parameters to Use When Calling Guest Binary** Press **Enter** to leave this field empty.
   - **Number of Instances of Guest Binary** Press **Enter** to accept the default setting of 1.

   After you respond to all the prompts, Yeoman creates a folder with the application name and generates an application package. It also creates two scripts, install.sh and uninstall.sh, which you can use for installing and uninstalling the application.

Note Yeoman does not define service endpoints. You'll need to add service endpoint configurations yourself. Furthermore, at the time of this writing, the generator doesn't give you any warnings if you've specified a wrong path for binary artifacts. You need to make sure you've entered the correct absolute path to your binary app artifacts.

Index

Numbers

A

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U