

2ND EDITION

# Practical Microservices with Dapr and .NET

A developer's guide to building cloud-native  
applications using the event-driven runtime



**DAVIDE BEDIN**

Foreword by Mark Russinovich, Azure CTO and Technical Fellow at Microsoft

# Practical Microservices with Dapr and .NET

A developer's guide to building cloud-native applications using the event-driven runtime

**Davide Bedin**



BIRMINGHAM—MUMBAI

# Practical Microservices with Dapr and .NET

Copyright © 2022 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Group Product Manager:** Gebin George

**Publishing Product Manager:** Sathyanarayanan Ellapulli

**Senior Editor:** Rounak Kulkarni

**Technical Editor:** Pradeep Sahu

**Copy Editor:** Safis Editing

**Project Coordinator:** Manisha Singh

**Proofreader:** Safis Editing

**Indexer:** Sejal Dsilva

**Production Designer:** Ponraj Dhandapani

**Developer Relations Marketing Executive:** Sonakshi Bubbar

First published: June 2019

Second edition: November 2022

Production reference: 1141022

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80324-812-7

[www.packt.com](http://www.packt.com)

*To my wife Silvia, and my sons, Abramo and Elia: nothing would have been possible without you.*

*– Davide Bedin*

# Foreword

It's been three years since November 2019, when Microsoft launched the initial public release of Dapr, a distributed application runtime, and two years since the first edition of this book was published.

Dapr started as an incubation called *Actions* in the Azure Office of the CTO in early 2019. Its goal was to simplify the development of distributed microservice-based cloud-native applications. I can say with pride that Dapr is now widely seen as doing just that. It's a **Cloud Native Computing Foundation** (CNCF) incubating project and is one of the very few that skipped sandbox stage when accepted to the CNCF. Dapr has over 1000 contributors, over 100 components, is in production at dozens of enterprises, and has even been integrated into cloud-native compute services by multiple cloud providers.

Dapr's novel innovation is that it implements a growing collection of abstract APIs, called *building blocks*, that it makes available to applications via a side-car HTTP or gRPC interface. The building blocks include service invocation for microservices to exchange synchronous requests, pub/sub for asynchronous message delivery, state store for saving data, input/output for integrating with external services, secret management, configuration management, and more.

Because applications use Dapr APIs and not service APIs directly, you can plug in a *component* (which is an implementation of a building block) without changing application code. Using a building block makes code portable across different environments with respect to the building block's functionality, including across multiple clouds. If you want to deploy code to Azure, for example, you can plug in the Event Hubs component into the pub/sub building block, then swap it out to Kinesis when you deploy the application to AWS. Not only that, but Dapr manages authentication, retries, throttling, and circuit breaking, and has built-in monitoring for all its APIs, insulating developers from having to worry about those mundane yet critical programming practices.

Unlike other distributed programming models, Dapr's side-car architecture enables code written in any language and the use of any framework to leverage Dapr's services. Furthermore, applications can use building blocks a la carte, or choose to use them fully. That makes Dapr incrementally adoptable so that you can *daprize* your existing projects at your own pace. In summary, Dapr is becoming the de facto cloud OS API surface.

In the two years since the first edition of this book was published, the book has helped thousands of .NET developers to jump start their use of Dapr. Since then, Dapr's capabilities have grown, and so has the number of building blocks. While nothing prevents you from using Dapr APIs directly via HTTP or gRPC, the .NET SDK has kept pace to make idiomatically native .NET interfaces.

Whether you're new to Dapr, or a Dapr expert and owner of the first edition of this book, this update is for you. There's no faster way for you to learn not just Dapr, but the tips and tricks that will enable you to focus on your .NET application's business logic and let Dapr, your cloud-native butler, take care of the rest.

Mark Russinovich  
Azure CTO and Technical Fellow  
Microsoft

# Contributors

## About the author

**Davide Bedin** is a cloud-native architecture enthusiast, with strong and relevant experience with cloud platforms. As the CTO of an ISV, Davide led its transformation process with the objective of creating new solutions based on the Microsoft Azure cloud. Davide particularly focused on the evolution of distributed computing to service-oriented architectures and, ultimately, microservices, spending most of his developer career creating web services. As a senior cloud solution architect at Microsoft, Davide is responsible for the guidance and support of enterprise customers in embracing the cloud paradigm, a key enabler of their digital transformation; lately, he has also been playing with Dapr.

*I thank Francesco Lana, Andrea Tosato, and Marco Zamana for their advice in creating the second edition of this book.*

*Also, I have been blessed with the wisdom of many great teachers during my career: there are too many to list them all here – you all know who you are.*

## About the reviewers

**Andrea Tosato** is a full-stack software engineer and architect on .NET applications. Andrea successfully develops .NET applications in various industries, sometimes facing complex technological challenges. He deals with desktop, web, and mobile development but with the arrival of the Azure cloud, it became his passion. In 2017, he co-founded Cloudgen Verona, a .NET community based in Verona, Italy, with his friend Marco Zamana. In 2019, he was named Microsoft MVP for the first time in the Azure category. Andrea graduated from the University of Pavia with a degree in computer engineering in 2008 and successfully completed his master's degree, also in computer engineering, from Modena in 2011. Andrea was born in 1986 in Verona, Italy, where he currently works as a remote worker. You can find Andrea on Twitter.

**Marco Zamana** lives in Verona in the magnificent hills of Valpolicella. He has a background as a software developer and solution architect. He was Microsoft's Most Valuable Professional for 3 years in the artificial intelligence category.

He currently works as a cloud solution architect engineering at Microsoft. He is the co-founder of Cloudgen Verona, a community that wants to discuss topics related to the cloud and, above all, Azure.



# Table of Contents

Preface	xv
---------	----

---

## Part 1: Introduction to Dapr

### 1

Introducing Dapr	3
Technical requirements	3
Understanding Dapr	4
What Dapr is not	6
Exploring Dapr	7
Setting up Dapr	9
Docker	10
Intalling the Dapr CLI	10
Installing .NET	10
Installing VS Code	11
Installing Windows Terminal	11
Installing Dapr in self-hosted mode	12
Installing Dapr on Kubernetes	13
Updating Dapr version	13
Building our first Dapr sample	14
Summary	18
Questions	19
Further reading	19

### 2

Debugging Dapr Solutions	21
Technical requirements	21
Configuring VS Code debug for Dapr	22
Attaching the debugger	22
Examining the debug configuration	24
Debugging a Dapr multi-project solution	28
Creating .NET solutions	29
Launching the configuration	30
Tasks	31
Launching debug sessions individually	32
Launching compound debug sessions	33
Using Tye with Dapr	35
Installing Tye	35

Using Tye	35	Questions	38
Summary	38	Further reading	38

## 3

---

### Microservices Architecture with Dapr 39

---

Introducing our sample solution	39	An example – sales microservices	47
Discovering microservices	40	Building microservices with Dapr	48
Service	41	Loosely coupled microservices	48
Autonomy	41	Autonomous microservices	48
Automated deployment	42	Observable microservices	48
Bounded context	42	Scalable microservices	49
Loose coupling	42	Event-driven microservices	49
Event-driven architecture	42	Stateless microservices	49
Observability	43	Summary	50
Sustainability	43	Questions	50
Adopting microservices patterns	44	Further reading	50
Building an e-commerce architecture	45		
Bounded contexts	46		

## Part 2: Building Microservices with Dapr

## 4

---

### Service-to-Service Invocation 53

---

Technical requirements	53	Implementing Dapr with an ASP.NET controller	61
Invoking services with Dapr	54	Creating a project for the Reservation service	63
Introducing service-to-service invocation	55	Preparing the debugging configuration	64
Name resolution	57	Implementing Dapr with an ASP.NET	
Resiliency	57	Minimal API	64
		Recap	68
Service invocation with the .NET SDK	60	Comparing HTTP and gRPC for Dapr	68
Creating a project for the Order service	60	gRPC in ASP.NET	68
Configuring Dapr in ASP.NET	61		

Creating a gRPC microservice	69	<b>Summary</b>	<b>76</b>
Winning latency with gRPC	76	<b>Questions</b>	<b>77</b>
		<b>Further reading</b>	<b>77</b>

## 5

### **Introducing State Management 79**

<b>Technical requirements</b>	<b>79</b>	Stateful reservation-service	86
<b>Managing state in Dapr</b>	<b>80</b>	Handling the Dapr state in ASP.NET	86
State, stateless, and stateful	80	<b>Using Azure Cosmos DB as a state store</b>	<b>89</b>
State stores in Dapr	80	Setting up Azure Cosmos DB	89
Transactions	81	Configuring the state store	90
Concurrency	82	Testing the state store	93
Consistency	82	Partitioning with Cosmos DB	95
Interaction with state stores	83	Wrapping up	97
Resiliency	84	<b>Summary</b>	<b>98</b>
<b>Stateful services in an e-commerce ordering system</b>	<b>85</b>		

## 6

### **Publish and Subscribe 99**

<b>Technical requirements</b>	<b>99</b>	Inspecting the messages	109
<b>Using the pub/sub pattern in Dapr</b>	<b>99</b>	<b>Implementing a saga pattern</b>	<b>111</b>
<b>Using Azure Service Bus (ASB) in Dapr</b>	<b>102</b>	Publishing messages to Dapr	113
Subscribing to a topic	104	Subscribing to a Dapr topic	114
Configuring a pub/sub component	106	Testing the saga pattern	115
Publishing to a topic	108	<b>Summary</b>	<b>117</b>

## 7

### **Resource Bindings 119**

<b>Technical requirements</b>	<b>119</b>	Configuring a cron input binding	121
<b>Learning how to use Dapr bindings</b>	<b>120</b>	Testing the cron binding	122

<b>Using Twilio output bindings in Dapr</b>	<b>122</b>	<b>Ingesting data with the Azure Event Hubs input binding</b>	<b>128</b>
Signing up for a Twilio trial	123	Creating an Azure Event Hubs binding	129
Configuring a Twilio output binding	124	Configuring the input binding	130
Signaling via the output binding	125	Implementing an Azure Event Hubs input binding	131
Verifying the notification	127	Producing events	132
		<b>Summary</b>	<b>134</b>

## 8

### Using Actors 135

<b>Technical requirements</b>	<b>135</b>	Resiliency	145
<b>Using actors in Dapr</b>	<b>136</b>	Lifetime	145
Introduction to the virtual actor pattern	136	<b>Implementing actors in an e-commerce reservation system</b>	<b>146</b>
Configuring a new state store	140	Preparing the Actor's projects	146
Verifying the configuration	141	Implementing the actor's model	147
<b>Actor concurrency, consistency, and lifetime</b>	<b>142</b>	Accessing actors from other Dapr applications	150
Placement service	142	Inspecting the actor's state	154
Concurrency and consistency	143	<b>Summary</b>	<b>155</b>

## Part 3: Deploying and Scaling Dapr Solutions

## 9

### Deploying to Kubernetes 159

<b>Technical requirements</b>	<b>159</b>	<b>Kubernetes</b>	<b>166</b>
<b>Setting up Kubernetes</b>	<b>160</b>	Building Docker images	167
Creating an Azure resource group	160	Pushing Docker images	170
Creating an AKS cluster	161	Managing secrets in Kubernetes	172
Connecting to the AKS cluster	161	Deploying applications	174
<b>Setting up Dapr on Kubernetes</b>	<b>162</b>	<b>Exposing Dapr applications to external clients</b>	<b>179</b>
<b>Deploying a Dapr application to</b>		<b>Summary</b>	<b>184</b>

## 10

### Exposing Dapr Applications 185

Technical requirements	185	Configuring Dapr policies with API management	195
Daprizing ingress controllers	186	Summary	203
Setting up API management on Kubernetes	189		

## 11

### Tracing Dapr Applications 205

Technical requirements	205	Analyzing metrics with Prometheus and Grafana	219
Observing applications in Dapr	206	Installing Prometheus	219
Tracing with Zipkin	208	Installing Grafana	220
Setting up Zipkin	208	Importing dashboards	221
Configuring tracing with Zipkin	210	Summary	223
Enabling tracing in Dapr	211		
Investigating with Zipkin	214		

## 12

### Load Testing and Scaling Dapr 225

Technical requirements	225	Load testing with Locust	234
Bash	226	Load testing Dapr	236
Python	226	Preparing the data via port-forwarding	236
Locust	227	Testing Locust locally	238
Kubernetes configuration	227	Locust on Azure Container Instances	240
Scaling Dapr on Kubernetes	227	Observing the Horizontal Pod Autoscaler	242
Replicas	229	Autoscaling with KEDA	245
Autoscale	230	Summary	246
Resource requests and limits	231		

13

<b>Leveraging Serverless Containers with Dapr</b>		<b>247</b>
Technical requirements	248	Exposing Azure container apps to external clients 259
Learning about the Azure Container Apps architecture	248	Observing Azure container apps 262
Understanding the importance of Azure Container Apps	250	<b>Autoscaling Azure Container Apps with KEDA</b> 267
Setting up Azure Container Apps	253	Learning about KEDA autoscalers 267
Deploying Dapr with Azure Container Apps	254	Applying KEDA to Azure container apps 268
Configuring Dapr components in Azure Container Apps	254	Testing KEDA with container apps 271
		<b>Summary</b> 274
<b>Assessments</b>		<b>275</b>
<b>Index</b>		<b>277</b>
<b>Other Books You May Enjoy</b>		<b>286</b>

# Preface

Practical microservices with **Distributed Application Runtime (Dapr)** and .NET helps you discover the powerful capabilities of Dapr by implementing a sample application with microservice architecture, using one of Dapr's many building blocks in each chapter of this book.

Over the last decade, there has been a huge shift from heavily coded monolithic applications to finer, self-contained microservices. Dapr helps developers build cloud-native applications by providing the building blocks as an easy-to-use API. It offers platform-agnostic features for running your applications on the public cloud, on-premises, and even on edge devices.

This book aims to familiarize you with microservice architecture while managing application complexities and getting into the nitty-gritty of Dapr in no time. You will also see how it combines the simplicity of its implementation with its openness to multiple languages and platforms. We will explore how Dapr's runtime, services, building blocks, and SDKs will help you simplify the creation of resilient and portable microservices.

Dapr provides an event-driven runtime that supports the essential features you need to build microservices, such as service invocation, state management, and publish and subscribe messaging. You'll explore all of these in addition to various other advanced features with this practical guide to learning about Dapr.

This book guides you in creating a sample application based on Dapr, which you'll then deploy to Kubernetes, and the multiple ways of exposing your application to clients that you can leverage, including integrating API Management with Dapr. In this operating environment, you'll learn how to monitor Dapr applications using Zipkin, Prometheus, and Grafana. After learning how to perform load-testing on Dapr applications in Kubernetes, we will explore how to leverage Dapr with a serverless container service.

By the end of this book, you'll be able to write microservices easily using your choice of language or framework by implementing the industry best practices to solve any problems related to distributed systems.

## Who this book is for

This book is for developers looking to explore microservices architectures and implement them in Dapr applications using .NET examples. Whether you are new to microservices or have knowledge of this architectural approach and want to get hands-on experience of using Dapr, you'll find this book useful. Familiarity with .NET will help you to understand the C# samples and code snippets used in the book.

## What this book covers

*Chapter 1, Introducing Dapr*, will introduce you to the basics of Dapr, briefly exposing the features that make Dapr interesting for new cloud-native applications, as well as for inserting microservices into existing applications.

*Chapter 2, Debugging Dapr Solutions*, will focus on how to set up your Dapr development environment in VS Code to locally debug simple Dapr solutions, as well as more complex ones.

*Chapter 3, Microservices Architecture with Dapr*, will discuss the relevance of microservices architectures and starts to explore how Dapr as a runtime can make it easier to adopt this style.

*Chapter 4, Service-to-Service Invocation*, will instruct you on how services can discover and invoke each other via the Dapr infrastructure. With examples, you will understand how to implement services and invoke them from other Dapr-aware components and Dapr-unaware or external clients.

*Chapter 5, Introducing State Management*, will illustrate how a Dapr solution can manage states with different store types. Managing states for services and actors is a centerpiece of Dapr.

*Chapter 6, Publish and Subscribe*, will teach you about publish and subscribe, which is the messaging pattern used by Dapr to enable decoupled interactions between components. Input bindings enable you to trigger your microservice using an incoming Twilio SMS or an Azure Service Bus message.

*Chapter 7, Resource Bindings*, will detail how input bindings in Dapr enable you to design event-driven microservices and invoke external resources via a pluggable configuration.

*Chapter 8, Using Actors*, will help you learn about the powerful virtual actor model provided by Dapr, how to leverage it in a microservices-style architecture, and the pros and cons of different approaches.

*Chapter 9, Deployment to Kubernetes*, will help distinguish the basic differences between local Standalone mode and Kubernetes mode in terms of their operations. Specifically using Azure Kubernetes Service, we will deploy a Dapr sample application composed of several microservices to Kubernetes.

*Chapter 10, Exposing Dapr Applications*, will explore how we can expose our Dapr applications to our end users. Specifically using Azure Kubernetes Service, we will expose our Dapr sample application on Kubernetes via NGINX and Azure API Management.

*Chapter 11, Tracing Dapr Applications*, will outline the observability options in Dapr by exploring how traces, logs, and metrics are emitted and can be collected in Dapr using Zipkin, Prometheus, and Grafana.

*Chapter 12, Load-Testing and Scaling Dapr*, will elaborate on how the scaling of Dapr services and actors works in Kubernetes, and by leveraging Locust, the reader will also learn how to load-test a Dapr solution by simulating user behaviors via the Locust testing tool.

*Chapter 13, Leveraging Serverless Containers with Dapr*, will guide you on how to benefit from Kubernetes without its complexities by adopting serverless containers with Azure. We will understand how to deploy a Dapr application to Azure Container Apps.

## To get the most out of this book

While the samples in the book have been written on Windows 10, the technology stack used is multiplatform: VS Code, .NET 6, Dapr, Kubernetes, and Locust all offer tools and libraries for multiple platforms.

On Windows 10, it is recommended to have WSL 2 installed and enable the WSL 2 engine in Docker.

For detailed instructions on how to set up your environment, please see the *Setting up Dapr* section in *Chapter 1, Introducing Dapr*.

**The samples and scripts in this book have been tested with Dapr version 1.8 and .NET 6.**

Software/hardware covered in the book	Operating system requirements
Docker Engine – latest version	Windows, any version supporting .NET 6
.NET 6	Windows, any version supporting .NET 6
Dapr 1.8 or later	Windows, any version supporting .NET 6
VS Code – latest version	Windows, any version supporting .NET 6
Azure CLI – latest version	Windows, any version supporting .NET 6
Python 3.8 or later	Windows, any version supporting .NET 6
Locust 2.10 or later	Windows, any version supporting .NET 6

Access to an Azure subscription is required, as the samples leverage many Azure services. Each chapter will give you instructions and direct you to the documentation for further information.

**If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.**

After reading this book, continue your learning by looking through the Dapr documentation at <https://docs.dapr.io/> and follow the community meetings at <https://github.com/dapr/community#community-meetings>; these are a great opportunity to learn, ask questions, and share your experience with Dapr.

## Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Practical-Microservices-with-Dapr-and-.NET-Second-Edition>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. Please refer to this graphic bundle for better readability of images. You can download it here: <https://packt.link/eSCK1>

## Conventions used

There are a number of text conventions used throughout this book.

**Code in text:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “The `--debug` \* option of the Tye command waits for VS Code debug sessions to be attached to each service.”

A block of code is set as follows:

```
"compounds":  
  [  
    {  
      "name": "webApi + webApi2 w/Dapr",  
      "configurations": [".NET Core Launch w/Dapr  
        (webapi)", ".NET Core Launch w/Dapr (webapi2)"]  
    }  
  ]
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
{  
  "name": ".NET Core Launch w/Dapr (webapi2)",  
  "type": "coreclr",  
  "request": "launch",  
  "preLaunchTask": "daprd-debug-webapi2",
```

---

Any command-line input or output is written as follows:

```
PS C:\> curl http://localhost:60151/v1.0/invoke/hello-world/  
method/hello  
Hello, World
```

**Bold:** Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “Once we activate debugging in VS Code by selecting **the .NET Core Launch (web) with Dapr** configuration, this is what happens.”

#### Tips or important notes

Appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** If you have questions about any aspect of this book, email us at [customercare@packtpub.com](mailto:customercare@packtpub.com) and mention the book title in the subject of your message.

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packtpub.com/support/errata](http://www.packtpub.com/support/errata) and fill in the form.

**Piracy:** If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packt.com](mailto:copyright@packt.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

## Share Your Thoughts

Once you've read *Practical Microservices with Dapr and .NET - Second Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

---

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/978-1-80324-812-7>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly



# Part 1: Introduction to Dapr

This first part of the book will give you a starting point on Dapr with an overview of what it is and its main features and components. By understanding how to debug with Dapr, you will gain confidence in exploring the building blocks.

This part contains the following chapters:

- *Chapter 1, Introducing Dapr*
- *Chapter 2, Debugging Dapr Solutions*
- *Chapter 3, Microservices Architecture with Dapr*



# Introducing Dapr

Welcome to this second edition of *Practical Microservices with Dapr and .NET! Distributed Application Runtime (Dapr)* evolved greatly after the version one release and the innovations in .NET 6 prompted a refreshed and enriched edition of the book.

This chapter will introduce you to the Dapr project to teach you the core concepts of its architecture and prepare you to develop with Dapr.

Dapr accelerates the development of new cloud-native applications and simplifies the adoption of a microservice architecture.

In this chapter, we are going to cover the following main topics:

- Understanding Dapr
- Exploring Dapr
- Setting up Dapr
- Building our first Dapr sample

Learning about these topics is important to establish a solid foundation for our future endeavors with Dapr and microservice architectures. These basic concepts will guide our learning throughout the rest of this book.

Our first steps into the project will start with exploring Dapr and understanding how it works.

## Technical requirements

The code for this sample can be found on GitHub at <https://github.com/PacktPublishing/Practical-Microservices-with-Dapr-and-.NET-Second-Edition/tree/main/chapter01>.

In this chapter, the working area for scripts and code is expected to be `<repository path>\chapter01\`. In our local environment, it is `C:\Repos\practical-dapr\chapter01`.

Please refer to the *Setting up Dapr* section for a complete guide to the tools needed to develop with Dapr and work with the samples.

## Understanding Dapr

Dapr is an event-driven, portable runtime created by Microsoft with an open source approach and it is a **Cloud Native Computing Foundation (CNCF)** incubated project.

Being event-driven (which is emphasized in the definition of Dapr) plays an important role in microservices; this is because an application can be designed to efficiently react to events from external systems or other parts of the solution and to produce events as well in order to inform other services of new facts or to continue processing elsewhere, or at a later stage.

Dapr is portable as it can run locally on your development machine in self-hosted mode; it can also be deployed to the edge, or it can run on Kubernetes.

The following diagram shows the many building blocks provided by Dapr:

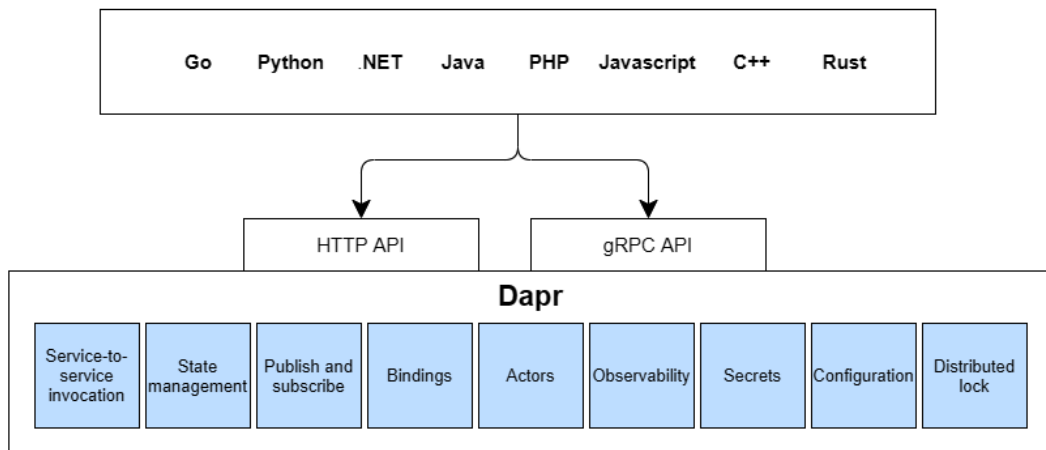


Figure 1.1 – Dapr architecture

Portability does also extend beyond the hosting environment—while Dapr is an initiative that was started by Microsoft, it can also run on Kubernetes on-premises or in the cloud with **Microsoft Azure**, **Amazon Web Services (AWS)**, **Google Cloud Platform (GCP)**, or any other cloud vendor.

Dapr has been built on the experience gained by Microsoft in developing hyperscale cloud-native applications. It has been inspired by the design of Orleans and Service Fabric, which in turn enables many Microsoft Azure cloud services to operate resiliently and at a large scale.

### A brief history of Dapr

Dapr was first released in October 2019, and you can find more information at <https://cloudblogs.microsoft.com/opensource/2019/10/16/announcing-dapr-open-source-project-build-microservice-applications/>.

Dapr adopted an open governance model early on in the initial development phase in September 2020; see the description at <https://blog.dapr.io/posts/2020/09/30/transitioning-the-dapr-project-to-open-governance/>.

Dapr reached the production-ready v1.0 release in February 2021; see <https://blog.dapr.io/posts/2021/02/17/announcing-dapr-v1.0/> for more details. In November 2021, Dapr joined CNCF as an incubated project; see the announcement at <https://blog.dapr.io/posts/2021/11/03/dapr-joins-cncf-as-an-incubating-project/>.

Dapr offers developers an approach to design the tools to build and the runtime to operate applications, based on a microservice architecture style.

Microservices offer a vast array of benefits balanced by increased complexities in team and product management, usually with a significant burden on the developer and the team in order to get started.

What if you could leverage a runtime such as Dapr to help you get through the common patterns you will likely need to adopt, and thus ease your operations?

The following figure shows the two Dapr hosting modes:

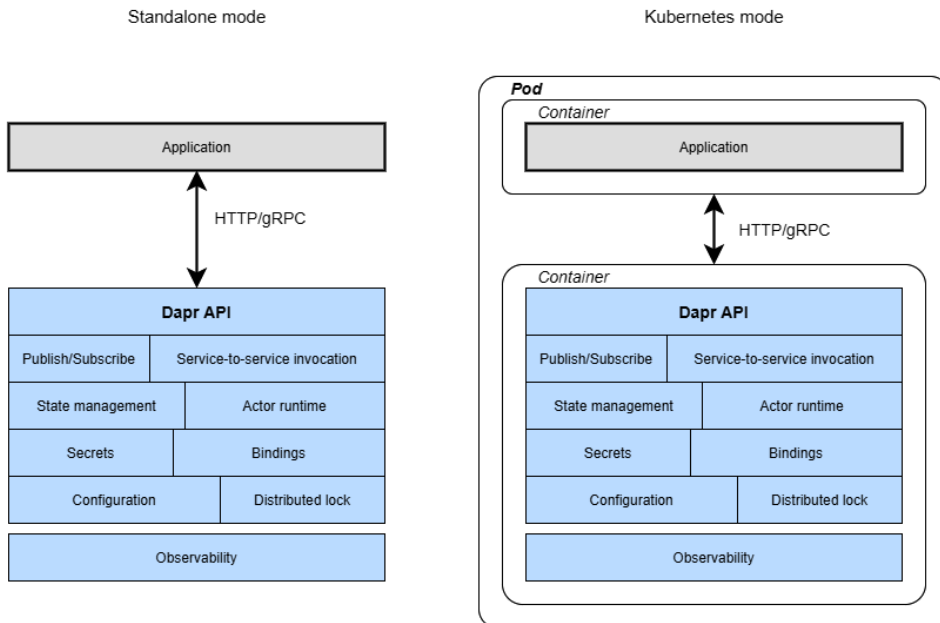


Figure 1.2 – Dapr sidecar

As depicted in *Figure 1.2*, the Dapr runtime operates in sidecar processes, lifting most of the complexity from the application to a separate environment, greatly simplifying development and operations as well. These sidecar processes are run locally in your development environment or as containers in a Pod on Kubernetes.

From an application perspective, Dapr is an **Application Programming Interface (API)** that can be directly reached via **HyperText Transfer Protocol (HTTP)**, **Remote Procedure Call (gRPC)** calls, or, even more simply, via any of the **Software Development Kits (SDKs)** available for .NET, Java, Go, Python, PHP, JavaScript, C++, and Rust languages.

As we will experience later, it is not necessary to adopt the Dapr SDK in your application; a request to a Dapr service can be as simple as an HTTP call to an endpoint, such as `http://localhost:3500/v1.0/invoke/<app-id>/method/<methodname>`. Nevertheless, using the SDK does provide many benefits if you are writing a Dapr service adopting the server extensions, interacting with Dapr via the client SDK, or leveraging the Dapr actor model with the Actor SDK.

You can learn more about SDKs and the supported languages in the Dapr docs at <https://docs.dapr.io/developing-applications/sdks/>.

Now that we have learned about the high-level architecture of Dapr, it is time to also clarify what Dapr is not.

## What Dapr is not

While we hope the overview of Dapr has informed and intrigued you enough to spend time on this book, when we have the chance to talk about Dapr, we often find ourselves in need of clarifying what Dapr is *not*. This makes it easier to eliminate any misconceptions we may have about what Dapr does, as follows:

- Dapr does *not* force the developer to embrace a programming model with strict rules and constraints. On the contrary, while the application developer is freed, by Dapr, of the many complexities of a microservice architecture, the developer is not mandated on how to write the application. As an example, the management of the connection to the database where the state is stored is a responsibility of Dapr and, as we will see in the following chapters, it is transparent to the microservice application code.
- Dapr is *not* a service mesh. While many similarities can be found in the general objectives of Dapr and service meshes, Dapr does provide these benefits at the application level, while a service mesh operates on the infrastructure. For instance, it is the developer's responsibility to decide how to handle errors Dapr might return if there is a conflict or an intermittent issue; whether it is adopting a retry policy as provided by the Dapr resiliency configurations, raising an error back to the client, or compensating the operation, these are explicit choices only the developer can make.

Dapr is designed to be integrated with service meshes such as Istio, which is out of the scope of this book.

- Dapr is *not* a Microsoft cloud service. It does help the developer build microservice applications in the cloud, and it surely provides many integrations with Azure cloud services, but it also has as many components for AWS, GCP, and other services. At the same time, Azure does offer rich support to Dapr in **Azure Kubernetes Service (AKS)** with the native extension, in **Azure API Management** with Dapr policies, and in **Azure Container Apps** with Dapr native integration.
- Dapr is *not* a .NET only technology. Dapr itself has been written in the Go language and any language can leverage it. SDKs for many languages are available but you can also choose to interact directly with the Dapr API without any additional library.

#### Important note

While this book is heavily skewed toward .NET, Dapr does provide the same benefits to Python developers (just as an example) as it provides SDKs for Dapr and Dapr Actor, with Kubernetes as the deployment target—Dapr welcomes all developers in a vendor-neutral and open approach.

We hope this perspective on the Dapr objectives and role will help you in properly adopting this technology. The next section will be dedicated to understanding the architecture of Dapr.

## Exploring Dapr

Dapr has been designed from the ground up as a set of pluggable building blocks—developers can create an application counting on the support of many facilities, while operators can adapt applications to the hosting environment by simply intervening in the configuration.

The following is a complete list of the tools and components of Dapr:

- **The Dapr command-line interface (CLI):** A cross-platform command-line tool to configure, manage, and monitor the Dapr environment. It is also the tool used to locally debug Dapr applications.
- **Dapr Helm charts:** It is worth mentioning that Helm charts are provided for a richer experience in installing and updating Dapr in a Kubernetes environment.
- **The Dapr API:** The API that defines how an application can interact with the Dapr runtime in order to leverage its building blocks.
- **The Dapr runtime:** This is the core of Dapr that implements the API. If you are curious, you can take a look at how it is developed in Go at Dapr's repository at <https://github.com/dapr/dapr>.
- **The Dapr host:** On your development machine, the host runs as a standalone process; in Kubernetes, it is a sidecar container in your application's pod.

- **The Dapr operator:** Specific to Kubernetes mode, the operator manages bindings and configurations.
- **The Dapr sidecar injector:** Once instructed via configuration in Kubernetes mode, this takes care of injecting the Dapr sidecar into your application pod.
- **The Dapr placement service:** This service has the objective of distributing (or placing) Actor instances across the Dapr pods.
- **Dapr Sentry:** A built-in **Certificate Authority (CA)** to issue and manage certificates used by Dapr to provide transparent **mutual Transport Layer Security (mTLS)**.

Dapr provides several building blocks that microservice application developers can adopt selectively, based on their needs, and they are as follows:

- **Service invocation:** Service-to-service invocation enables your code to call other services located in the same hosting environment while taking care of the retry policy.

This building block is presented in more detail in *Chapter 4, Service-to-Service Invocation*.

- **State management:** This is to efficiently manage the application state as a simple **Key-Value Pair (KVP)**, relieving your stateful or stateless services of the need to support different backends. Dapr provides many state stores, which include Redis, Azure Cosmos DB, Azure SQL Server, and PostgreSQL, which can be plugged in via configuration.

You can learn about this building block in *Chapter 5, Introducing State Management*.

- **Publish and subscribe (pub/sub) messaging:** The pub/sub pattern enables decoupled communication between microservices by exchanging messages, counting on the presence of a service bus, which can route messages between producers and consumers.

A discussion of this building block is presented in *Chapter 6, Publish and Subscribe*.

- **Resource bindings:** This is where the event-driven nature of Dapr shines. With bindings, your application can be triggered by a **Short Message Service (SMS)** message sent via Twilio (just one of the popular services in the area of communication API).

This building block is presented in more detail in *Chapter 7, Resource Bindings*.

- **Actors:** The actor pattern aims to simplify highly concurrent scenarios by splitting the overall request load between a large number of computation units (the actors), which take care of the job in their smaller, but independent, scope by processing requests to a single actor one at a time. Dapr provides great benefits in this space.

You can learn about this building block in *Chapter 8, Using Actors*.

- **Observability:** Dapr enables the developer and operator to observe the behavior of the system services and applications without having to instrument them.

This building block is presented in more detail in *Chapter 11, Tracing Dapr Applications*.

- **Secrets:** It is a healthy practice to keep secrets segregated from the code. Dapr enables you to store secrets and to reference these from other components, in Kubernetes or Azure Key Vault, among many options.
- **Configuration:** Introduced in Dapr version 1.8 in Alpha state, this building block addresses the common need to retrieve the configuration data needed by an application.
- **Distributed lock:** Introduced with Dapr version 1.8 in Alpha state, it provides a powerful lease-based mechanism to manage mutually exclusive access to a named lock. The lock can be used by the application to assure exclusive access to a resource by many concurrent instances.

After learning about Dapr architecture and components, and before we can start using them, we need to set up Dapr in our development environment, which will be the topic of the next section.

## Setting up Dapr

Dapr is a runtime for every platform and every language. The focus of this book is on C# in .NET, used with **Visual Studio Code (VS Code)**. The code snippets in the book can be appreciated by developers from any background, but nevertheless, you will get the most out of it from a .NET perspective.

The development environment we use is Windows, as you will be able to tell from the screenshots we use in the book. While the CLI, configuration, and files will be the same, if you need more details on how to perform a particular action on Linux or a macOS development machine, we encourage you to check the Dapr documentation at <https://docs.dapr.io/>.

### Dapr roadmap

The Dapr runtime reached the v1.0 production-ready release in February 2021, as announced in the Dapr blog at <https://blog.dapr.io/posts/2021/02/17/announcing-dapr-v1.0/>, and five new minor versions have been released during 2021. You can learn more about the Dapr roadmap at <https://docs.dapr.io/contributing/roadmap/>.

The samples and scripts in this book have been updated and tested with v1.8 of Dapr.

In this book, we will also leverage several services on the Azure cloud platform (<https://azure.microsoft.com/en-us/explore/>), whether as a platform to execute Dapr applications, for exchanging messages via Azure, or for storing data.

Access to an Azure subscription is required. Each chapter will give you instructions and direct you to documentation for further information.

Next, we will accomplish the following steps:

- Configuring Docker
- Installing the Dapr CLI
- Installing .NET
- Installing VS Code
- Installing Windows Terminal
- Installing Dapr in self-hosted mode
- Installing Dapr on Kubernetes

Let's start with Docker.

## Docker

Dapr requires Docker to be present locally in your development environment; therefore, make sure you have it installed. If your development machine is Windows, Docker must be running in Linux container mode.

You can find detailed instructions for running Docker at <https://docs.docker.com/install/>.

## Intalling the Dapr CLI

We will immediately start working with Dapr; therefore, you need to install all the necessary tools. The Dapr runtime and its tools can be found at <https://github.com/dapr/cli>.

On Windows, execute the following command to install the CLI in the `c:/dapr` directory and add it to the user PATH environment variable so that the tools can be easily found from the command line:

```
powershell -Command "iwr -useb https://raw.githubusercontent.com/dapr/cli/master/install/install.ps1 | iex"
```

For more details on the Dapr CLI, please refer to <https://docs.dapr.io/getting-started/install-dapr-cli/>.

We still need to initialize Dapr on the development machine, which we will do in the *Installing Dapr in self-hosted mode* section of this chapter.

## Installing .NET

To install .NET 6, please refer to <https://dotnet.microsoft.com/download> for the link to the latest binaries.

.NET 6 is a **Long-Term Support (LTS)** version of .NET, which gets free support and patches for 3 years. Refer to <https://dotnet.microsoft.com/en-us/platform/support/policy> for more details on the .NET support policy.

On a development machine, it makes sense to install the full SDK, which includes the runtime. Once the install is complete, open a new command prompt and run the `dotnet --info` command. You should see the following output:

```
PS C:\Repos\dapr-samples\chapter01> dotnet --info
.NET SDK (reflecting any global.json):
  Version:      6.0.101
  Commit:       ef49f6213a
Runtime Environment:
  OS Name:      Windows
  OS Version:   10.0.22000
  OS Platform:  Windows
  RID:          win10-x64
  Base Path:    C:\Program Files\dotnet\sdk\6.0.101\
Host (useful for support):
  Version: 6.0.1
  Commit:  3a25a7f1cc ...
```

This proves that .NET has been recognized and the framework is working fine.

## Installing VS Code

VS Code is a great multiplatform source code editor by Microsoft. You can install it for free by following the instructions at <https://code.visualstudio.com/docs/setup/windows>.

### *The Dapr extension*

Dapr has an extension for VS Code that helps with navigating the Dapr local environment and eases the debugging configuration—we highly recommend it. Please follow the instructions at <https://docs.dapr.io/developing-applications/ides/vscode/>.

## Installing Windows Terminal

We really love the new Windows Terminal (<https://aka.ms/terminal>) for its ease of use and configurability. In the following chapters, we will often have to run multiple commands and tools in parallel. Therefore, the tabs feature of Windows Terminal is just one of the reasons why we suggest you adopt it too.

## Installing Dapr in self-hosted mode

Dapr can be initialized in two modes: **self-hosted** (or standalone) and **Kubernetes**.

As it is intended to be used for a development environment, the self-hosted mode locally installs Redis, the Dapr placement services, and Zipkin. The following command initializes Dapr in your local environment:

```
dapr init
```

The Dapr binaries and default components to leverage Redis are by default positioned in the %USERPROFILE%\ .dapr\ folder.

In a local development environment, it might happen that the ports Dapr might intend to use for Redis, for example, are already in use. In this case, you should identify which processes or containers are using the ports and change them accordingly.

Once you launch the `init` command, the following is the output you should expect:

```
PS C:\Repos\practical-dapr\chapter01> dapr init
Making the jump to hyperspace...
Installing runtime version 1.8.4
Downloading binaries and setting up components...
Downloaded binaries and completed components set up.
daprd binary has been installed to C:\Users\dabedin\.dapr\bin.
dapr_placement container is running.
dapr_redis container is running.
dapr_zipkin container is running.
Use `docker ps` to check running containers.
Success! Dapr is up and running. To get started, go here:
https://aka.ms/dapr-getting-started
```

To check your newly initialized Dapr environment, you can use `docker ps` as follows:

```
PS C:\Repos\practical-dapr\chapter01> docker ps --format "{{.
Image}}" - "{{.Ports}}" - "{{.Names}}}"
daprio/dapr:1.8.4 - 0.0.0.0:6050->50005/tcp, :::6050->50005/tcp
- dapr_placement
openzipkin/zipkin - 9410/tcp, 0.0.0.0:9411->9411/tcp, :::9411-
>9411/tcp - dapr_zipkin
redis - 0.0.0.0:6379->6379/tcp, :::6379->6379/tcp - dapr_redis
```

The output shows the Docker container for Dapr running on our machine.

## Installing Dapr on Kubernetes

Dapr is specifically intended to be executed on Kubernetes. From your development machine on which you have the Dapr CLI installed, you can set up Dapr on the Kubernetes cluster currently configured as follows:

```
dapr init -k
```

Alternatively, you can install Dapr on Kubernetes with a Helm v3 chart. You can find more details at <https://docs.dapr.io/getting-started/install-dapr-kubernetes/#install-with-helm-advanced>.

### Important note

If you intend to define a **Continuous Integration/Continuous Deployment (CI/CD)** pipeline that takes care of the Dapr installation on the Kubernetes cluster too, this can also work, although it is out of scope for the present setup.

To verify the installation was successfully completed, execute the following command:

```
kubectl get pods --namespace dapr-system
```

The command should display the pods in the `dapr-system` namespace.

## Updating Dapr version

On a development Windows machine on which a previous version of Dapr was already present, the CLI can be updated by simply re-installing with the command we saw in a previous section.

As described in <https://docs.dapr.io/operations/hosting/self-hosted/self-hosted-upgrade/>, on a Windows development machine on which a previous version of Dapr was already present, you must uninstall Dapr first as follows:

```
PS C:\Repos\practical-dapr\chapter01> dapr uninstall --all
```

With the CLI updated and Dapr uninstalled, we can repeat the Dapr installation as follows:

```
PS C:\Repos\practical-dapr\chapter01> dapr init
```

After we execute `dapr init`, checking the Dapr version, we can see it has now moved forward from 1.0 to 1.1 for both the CLI and the runtime, as illustrated in the following code snippet:

```
PS C:\Repos\practical-dapr\chapter01> dapr --version
CLI version: 1.8.1
Runtime version: 1.8.4
```

Our Dapr test environment is up and running. We are now ready to try it with our first sample.

## Building our first Dapr sample

It is time to see Dapr in action. We are going to build a web API that returns a `hello world` message. We chose to base all our samples in the `C:\Repos\practical-dapr\` folder, and we created a `C:\Repos\practical-dapr\chapter01` folder for this first sample. We'll take the following steps:

1. Let's start by creating a Web API ASP.NET project as follows:

```
PS C:\Repos\practical-dapr\chapter01> dotnet new webapi
-o dapr.microservice.webapi
```

2. Then, we add the reference to the Dapr SDK for ASP.NET. The current version is `1.5.0`. You can look for the package versions on NuGet at <https://www.nuget.org/packages/Dapr.Actors.AspNetCore> with the `dotnet add package` command, as illustrated in the following code snippet:

```
PS C:\Repos\practical-dapr\chapter01> dotnet add package
Dapr.AspNetCore --version 1.8.0
```

3. We need to apply some changes to the template we used to create the project. These are going to be much easier to do via VS Code—with the `<directory>\code .` command, we open it in the scope of the project folder.
4. To support Dapr in ASP.NET 6 and leverage minimal hosting and global usings, we made a few changes to the code in `Program.cs`. We changed the `builder.Services.AddControllers()` method to `builder.Services.AddControllers().AddDapr()`.

We also added `app.MapSubscribeHandler()`. While this is not necessary for our sample, as we will not use the pub/sub features of Dapr, it is better to have it in mind as the base set of changes you need to apply to a default ASP.NET project.

Finally, in order to simplify the code, we commented `app.UseHttpsRedirection()`.

The following is the modified code of the `Program.cs` class:

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddControllers().AddDapr();
// Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
var app = builder.Build();
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
//app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.MapSubscribeHandler();
app.Run();
```

In the preceding code block, we instructed Dapr to leverage the **Model-View-Controller (MVC)** pattern in ASP.NET 6.

5. Finally, we added a controller named `HelloWorldController` as illustrated in the following code snippet:

```
using Microsoft.AspNetCore.Mvc;
namespace dapr.microservice.webapi.Controllers;
[ApiController]
[Route("[controller]")]
public class HelloController : ControllerBase
{
    private readonly ILogger<HelloController> _logger;
    public HelloController(ILogger<HelloController>
        logger)
    {
        _logger = logger;
    }
}
```

```
    }  
    [HttpGet()]  
    public ActionResult<string> Get()  
    {  
        Console.WriteLine("Hello, World.");  
        return "Hello, World";  
    }  
}
```

In the preceding code snippet, you can see `[Route]` and `[HttpGet]`. These ASP.NET attributes are evaluated by the routing to identify the method name.

6. In order to run a Dapr application, you use the following command structure:

```
dapr run --app-id <your app id> --app-port <port of the  
application> --dapr-http-port <port in Dapr> dotnet run
```

We left the ASP.NET default port as 5000 but we changed the Dapr HTTP port to 5010. The following command line launches the Dapr application:

```
PS C:\Repos\practical-dapr\chapter01\dapr.microservice.  
webapi> dapr run --app-id hello-world --app-port 5000  
--dapr-http-port 5010 dotnet run  
Starting Dapr with id hello-world. HTTP Port: 5010. gRPC  
Port: 52443
```

The initial message informs you that Dapr is going to use port 5010 for HTTP as specified, while for gRPC, it is going to auto-select an available port.

The log from Dapr is full of information. To confirm your application is running correctly in the context of the Dapr runtime, you can look for the following code:

```
Updating metadata for app command: dotnet run  
You're up and running! Both Dapr and your app logs will  
appear here.
```

At this stage, ASP.NET is responding locally on port 5000 and Dapr is responding on port 5010. In order to test Dapr, let's invoke a `curl` command as follows, and using the browser is equally fine:

```
PS C:\Repos\practical-dapr\chapter01> curl http://  
localhost:5010/v1.0/invoke/hello-world/method/hello  
Hello, World
```

This exciting response has been returned by Dapr, which passed our (the client's) initial request to the ASP.NET Web API framework. You should also see that the same result logged as `Console.WriteLine` sends its output to the Dapr window as follows:

```
== APP == Hello, World.
```

7. From another window, let's verify our Dapr service details. Instead of using the `dapr list` command, let's open the Dapr dashboard as follows:

```
PS C:\Windows\System32> dapr dashboard
Dapr Dashboard running on http://localhost:8080
```

We can open the dashboard by navigating to `http://localhost:8080` to reveal the following screen:

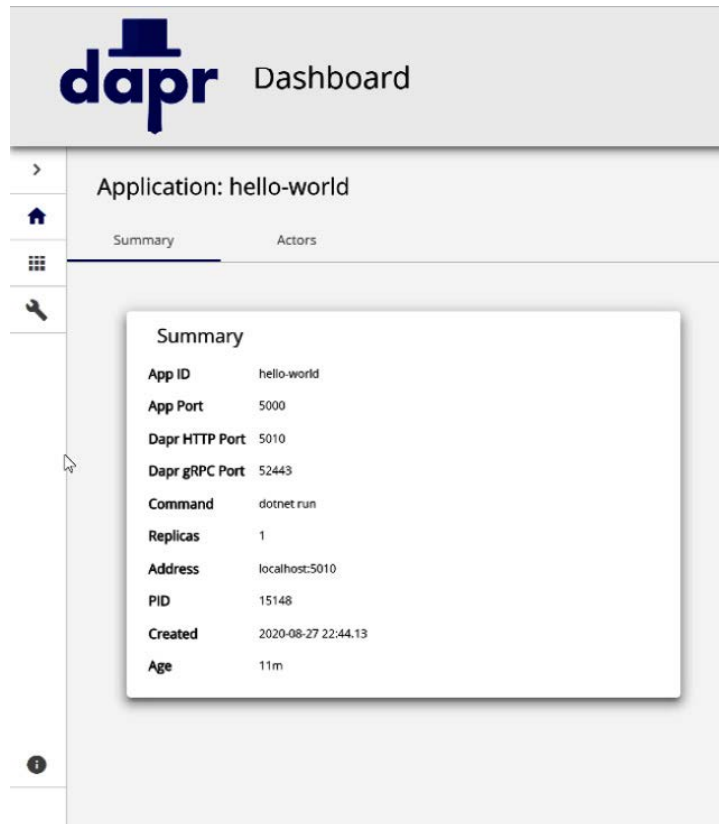


Figure 1.3 – Dapr dashboard application

The Dapr dashboard shown in *Figure 1.3* illustrates the details of our `hello-world` application.

In this case, the Dapr dashboard shows only this sample application we are running on the development machine. In a Kubernetes environment, it would show all the microservices running, along with the other components.

The Dapr dashboard also displays the configured components in the hosting environment, as we can see in the following screenshot:

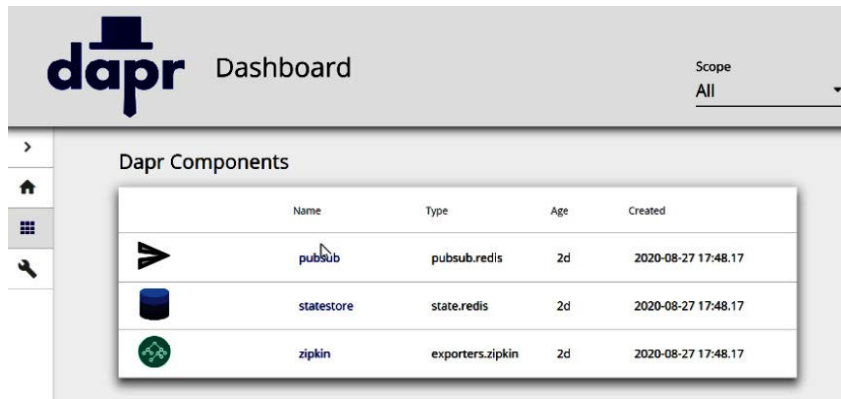


Figure 1.4 – Dapr dashboard components

In *Figure 1.4*, the Dapr dashboard shows us that the local installation of Redis is configured as state store and pub/sub components, in addition to the deployment of Zipkin.

This ends our introductory section, where we were able to build our first Dapr sample.

## Summary

In this chapter, you have learned about the Dapr project, its components, building blocks, and the sidecar approach. All of these concepts will be explored individually in further depth in the following chapters. You are now able to set up Dapr on your local development machine and prepare all the necessary tools to make this experience easier.

You have also learned how to create a simple ASP.NET project and how to configure and check Dapr, and we have had a glimpse of the Dapr dashboard where we can gain a complete and immediate view of the Dapr environment.

In the next chapter, we will use the newly created environment to learn how to debug Dapr.

## Questions

1. Which building blocks does Dapr provide?
2. What is the relationship between the Dapr CLI and Dapr runtime?
3. How can you install Dapr on your local development environment?
4. Which alternative methods can you follow to install Dapr in Kubernetes?

## Further reading

- *Overview of Dapr*: <https://docs.dapr.io/concepts/overview/>
- *Getting Started with Dapr*: <https://docs.dapr.io/getting-started/>
- *Roadmap of Dapr*: <https://docs.dapr.io/contributing/roadmap/>



# Debugging Dapr Solutions

In this chapter, you will learn how to set up your local development environment in **Visual Studio Code (VS Code)** to locally debug simple **Distributed Application Runtime (Dapr)** solutions, as well as more complex ones.

The base concepts of Dapr execution are presented with several different approaches—with the **command-line interface (CLI)**, with a VS Code debug session, and with Tye. Depending on your preferences, you will choose the one that suits you most and adopt it throughout the rest of the book.

This chapter covers the following topics:

- Configuring VS Code debug for Dapr
- Debugging a Dapr multi-project solution
- Using Tye with Dapr

In learning, there is no substitute for practice. Dapr is no exception, and to practice it, we will launch one Dapr application (or many) to investigate how it behaves—the sooner we are able to debug it, the better. We will start by configuring in VS Code.

## Technical requirements

The code for this chapter can be found on GitHub at <https://github.com/PacktPublishing/Practical-Microservices-with-Dapr-and-.NET-Second-Edition/tree/main/chapter02>.

In this chapter, the working area for scripts and code is expected to be `<repository path>\chapter02\`. In my local environment, it is `C:\Repos\practical-dapr\chapter02`.

Please refer to the *Setting up Dapr* section in *Chapter 1, Introducing Dapr*, for a complete guide to the tools needed to develop with Dapr and work with the samples.

## Configuring VS Code debug for Dapr

Throughout this book, we will leverage VS Code in exploring Dapr via several examples. Before we delve into the detailed features of Dapr, we need to understand how VS Code, the multi-platform code editor, can be configured to help us debug our sample code.

For an extended guide on how to set up Dapr debugging in VS Code, see the documentation at <https://docs.dapr.io/developing-applications/ides/vscode/vscode-how-to-debug-multiple-dapr-apps/>.

The following sections will guide us in configuring debugging in VS Code on a copy of our hello-world sample from the previous chapter.

### Attaching the debugger

In the working area of the chapter, you will find the samples from *Chapter 1, Introducing Dapr*. From Command Prompt, you can launch a sample via the Dapr CLI with the `dotnet run` command to launch the Web API project, previously modified to accommodate Dapr, as follows:

```
PS C:\Repos\practical-dapr\chapter02\sample.microservice.webapi> dapr run --app-id hello-world --app-port 5000 --dapr-http-port 5010 dotnet run
```

You can expect this as the output:

```
Starting Dapr with id hello-world. HTTP Port: 5010. gRPC Port: 57138
...
Updating metadata for app command: dotnet run
You're up and running! Both Dapr and your app logs will appear here.
```

Without any additional configuration, other than the default provided by VS Code, we can attach the .NET running code to debug our Dapr sample service.

As we are going to change the debug configurations later, I suggest that we grow accustomed to it.

Here is the default content of the `launch.json` file for a .NET Web API project—at this stage, we will focus on the .NET Core Attach configuration:

```
{
  "version": "0.2.0",
  "configurations": [
```

```

{
  "name": ".NET Core Launch (web)",
  "type": "coreclr",
  "request": "launch",
  "preLaunchTask": "build",
  //
  "program": "${workspaceFolder}/bin/Debug/net6.0
    /sample.microservice.webapi.dll",
  "args": [],
  "cwd": "${workspaceFolder}",
  "stopAtEntry": false,
  /* Enable launching a web browser when ASP.NET
  Core starts. For more information: https://aka.ms/
  VSCode-CS-LaunchJson-WebBrowser */
  "serverReadyAction": {
    "action": "openExternally",
    "pattern": "\\bNow listening
      on:\\s+(https?://\\S+) "
  },
  "env": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  },
  "sourceFileMap": {
    "/Views": "${workspaceFolder}/Views"
  }
},
{
  "name": ".NET Core Attach",
  "type": "coreclr",
  "request": "attach",
  "processId": "${command:pickProcess}"
}
]
}

```

In the preceding configuration, there is nothing special, other than asking the end user the process to which to attach the VS Code debugging experience.

**Important note: configuration default naming**

You will often read the name *.NET Core* in the configuration files prepared for you in VS Code. It is nothing more than a residue from the past version of .NET, and is not an indication of the .NET version used throughout the project and debug files.

With our Dapr hello-world service running in a Dapr runtime process (`daprd.exe`) launched via the Dapr CLI, by starting the *.NET Core Attach* VS Code debugging configuration, we should look for the *.NET Web API* server process instead. In this context, it can be found as `sample.microservice.webapi.exe`, as you can see from the following screenshot:

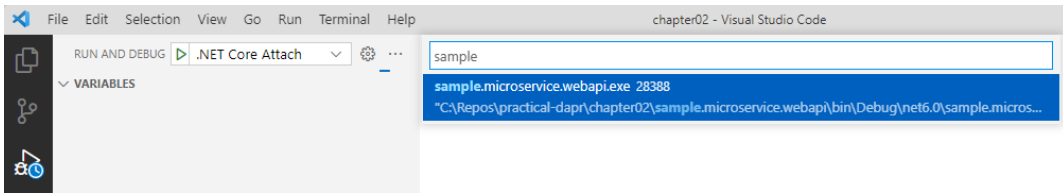


Figure 2.1 – Attaching debug to process in VS Code

Once we have selected the right process to attach the debugger to, we should be able to set breakpoints in VS Code and properly debug our code.

This was an easy approach to start with, although it might prove less than optimal with frequent debugging or more complex projects.

Next, we'll examine the configuration in more detail.

## Examining the debug configuration

The next objective is to instruct VS Code to combine both steps: launching Dapr via the CLI and attaching the debugger.

**Tip**

You can activate the Command Palette in VS Code with the *Ctrl + Shift + P* keys. For more details on the tool's **user interface (UI)**, see <https://code.visualstudio.com/docs/getstarted/userinterface>.

Instead of manually configuring `launch.json`, we can leverage the Dapr extension for VS Code to scaffold the configuration for us.

In our hello-world sample, let's open the Command Palette and look for Dapr tasks. Among these, we can find **Dapr: Scaffold Dapr Tasks**, as illustrated in the following screenshot:

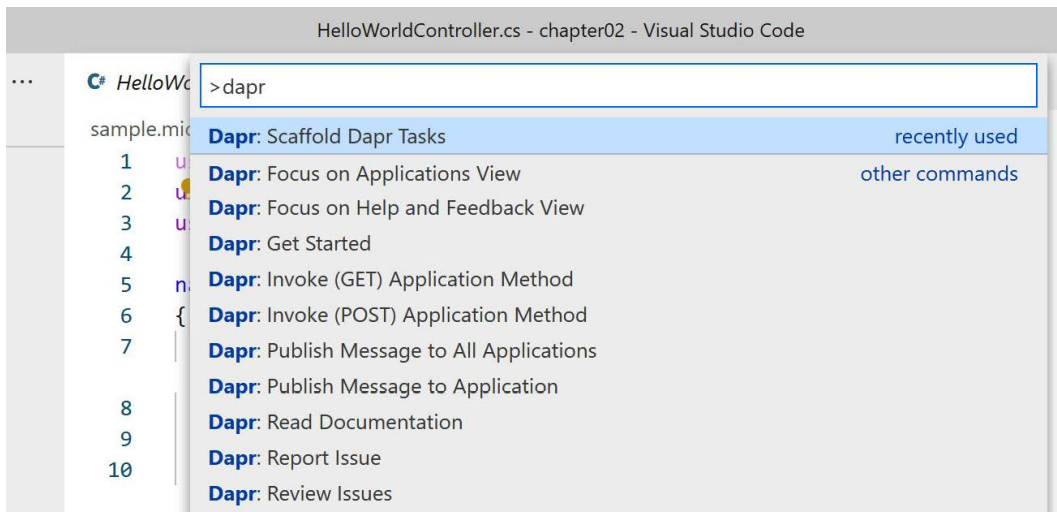


Figure 2.2 – Tasks in the Dapr extension for VS Code

By selecting it, we get asked which launch base configuration to derive from: the default **.NET Core Launch (web)** is the one to choose.

To stay consistent with our direct usage of the Dapr CLI, we set `hello-world` as the app **identifier (ID)** and leave the port at the default `app-port 5000`. The following extract of `launch.json` displays the relevant changes:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": ".NET Core Launch (web)",
      ... omitted ...
    },
    {
      "name": ".NET Core Attach",
      ... omitted ...
    },
    {
      "name": ".NET Core Launch (web) with Dapr",
      "type": "coreclr",
      "request": "launch",
      "preLaunchTask": "daprd-debug",
    }
  ]
}
```

```

        "program": "${workspaceFolder}/bin/Debug/net6.0
            /sample.microservice.webapi.dll",
        "args": [],
        "cwd": "${workspaceFolder}",
        "stopAtEntry": false,
        "serverReadyAction": {
            "action": "openExternally",
            "pattern": "\\bNow listening
                on:\\s+(https?://\\S+) "
        },
        "env": {
            "ASPNETCORE_ENVIRONMENT": "Development"
        },
        "sourceFileMap": {
            "/Views": "${workspaceFolder}/Views"
        },
        "postDebugTask": "daprd-down"
    }
]
}

```

A `tasks.json` file has also been prepared by the Dapr extension task, and this can be viewed here:

```

{
    "version": "2.0.0",
    "tasks": [
        ... omitted ...
        {
            "appId": "hello-world",
            "appPort": 5000,
            "label": "daprd-debug",
            "type": "daprd",
            "dependsOn": "build"
        },
        {
            "appId": "hello-world",
            "label": "daprd-down",

```

```
        "type": "daprd-down"
      }
    ]
  }
}
```

Once we activate debugging in VS Code by selecting the **.NET Core Launch (web) with Dapr** configuration, this is what happens, in order:

1. A "daprd-debug" task is invoked.
2. This task has a dependency on the "build" task, which, as the name implies, builds the .NET project.
3. The newly built .NET Web API project is executed.
4. The task has the type "daprd": the Dapr debugger is invoked with the configured settings.
5. Once we are done with debugging, the "daprd-down" task is invoked to stop the Dapr service.

#### Tip

Keep in mind that the VS Code debug configuration is an alternative approach to launching via the Dapr CLI: on a development environment, you cannot have multiple processes trying to run the same application ID with port numbers already in use.

I recommend that you explicitly edit the task in `task.json` to accommodate the local development environment needs.

To match the `dapr run -app-id hello-world -app-port 5000 -daprd-http-port 5010 dotnet run` Dapr CLI syntax we've used so far, `appPort` is the port used by the .NET Web API application, while `httpPort` and `grpcPort` are configured to expose Dapr endpoints. You can see these ports in the following code snippet:

```
{
  "appId": "hello-world",
  "appPort": 5000,
  "httpPort": 5010,
  "grpcPort": 50010,
  "label": "daprd-debug",
  "type": "daprd",
  "dependsOn": "build"
}
```

If we launch the VS Code debug configuration, here is the terminal output, showing the expected steps:

```
> Executing task: C:\Program Files\dotnet\dotnet.exe
build C:\Repos\practical-dapr\chapter02\
sample.microservice.webapi/sample.microservice.
webapi.csproj /property:GenerateFullPaths=true/
consoleloggerparameters:NoSummary <
Microsoft (R) Build Engine version 17.0.0+c9eb9dd64 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.
  Restore completed in 49,91 ms for C:\Repos\practical-dapr\
chapter02\sample.microservice.webapi\sample.microservice.
webapi.csproj.
  sample.microservice.webapi -> C:\Repos\practical-dapr\
chapter02\sample.microservice.webapi\bin\Debug\net6.0\sample.
microservice.webapi.dll
Terminal will be reused by tasks, press any key to close it.
> Executing task: daprd-debug <
> Executing command: daprd -app-id "hello-world" -app-port
"5000" -dapr-grpc-port "50010" -dapr-http-port "5010" -
placement-address "localhost:50005" <
```

The latest command shows how the debug configuration is leveraging the Dapr CLI with the exact ports and settings we specified for `task.json`.

By reaching Dapr at `http://localhost:5010/v1.0/invoke/hello-world/method/hello`, we received the expected results, this time in an integrated debugging experience inside of VS Code.

We have now successfully configured VS Code to be able to rapidly build our .NET project, start Dapr with our configuration, and clean up at the end. So, we'll do all that in the coming sections.

## Debugging a Dapr multi-project solution

In this section, we will configure debugging for multiple ASP.NET projects. In most cases, a .NET solution is developed in different projects, each representing a microservice or another component of the overall architecture. This is even more true in Dapr, which emphasizes and facilitates the development of microservices.

In this context, we leverage the VS Code capability to debug multiple projects at once. For more details, see the documentation at [https://code.visualstudio.com/docs/editor/debugging#\\_multitarget-debugging](https://code.visualstudio.com/docs/editor/debugging#_multitarget-debugging).

When we previously needed to launch with only one project, we leveraged a *scaffold* task in the VS Code Dapr extension to add Dapr support in the `launch.json` and `task.json` files.

This time, we will create a new .NET project and include both in a solution. Afterward, we will manually edit the debug configuration files.

## Creating .NET solutions

We previously provisioned our first .NET project. To test the multi-project debug configuration, we'll add a *second* Dapr project to our environment—in a great feat of imagination, let's name it `sample.microservice.webapi2`, as follows:

```
PS C:\Repos\practical-dapr\chapter02\sample.microservice.webapi> cd ..
PS C:\Repos\practical-dapr\chapter02> dotnet new webapi -o sample.microservice.webapi2
The template "ASP.NET Core Web API" was created successfully.
Processing post-creation actions...
Running 'dotnet restore' on sample.microservice.webapi2\sample.microservice.webapi2.csproj...
Restore completed in 152,92 ms for C:\Repos\practical-dapr\chapter02\sample.microservice.webapi2\sample.microservice.webapi2.csproj.
Restore succeeded.
```

For brevity, the newly created .NET project should receive the same changes we applied to our first sample in the *Building our first Dapr sample* section of *Chapter 1, Introducing Dapr*.

With two ASP.NET projects, we could use a solution file to combine them—the .NET CLI has the ability to create an empty `.sln` file and add projects to it, as we can see here:

```
PS C:\Repos\practical-dapr\chapter02> dotnet new sln
The template "Solution File" was created successfully.
PS C:\Repos\practical-dapr\chapter02> dotnet sln add c:\Repos\practical-dapr\chapter02\sample.microservice.webapi\sample.microservice.webapi.csproj
```

The solution file is ready; we can now prepare the debugging configuration.

## Launching the configuration

As we intend to debug `sample.microservice.webapi` (our original project) and `sample.microservice.webapi2` at the same time, the two .NET projects and the Dapr application need to be running at the same time.

It is required that each ASP.NET project is hosted on a different port and that each Dapr application has a different name and a unique **HyperText Transfer Protocol (HTTP)**, **gRPC Remote Procedure Call (gRPC)**, and metrics port, and should refer to the ASP.NET port as the Dapr app port.

The following code is a portion of the `launch.json` file:

```
... omitted ...
  "configurations": [
    {
      "name": ".NET Core Launch w/Dapr (webapi)",
      ... omitted ...
    },
    {
      "name": ".NET Core Launch w/Dapr (webapi2)",
      "type": "coreclr",
      "request": "launch",
      "preLaunchTask": "daprd-debug-webapi2",
      "program": "${workspaceFolder}/sample
        .microservice.webapi2/bin/Debug
        /net6.0/sample.microservice.webapi2.dll",
      "args": [],
      "cwd": "${workspaceFolder}
        /sample.microservice.webapi2",
      "stopAtEntry": false,
      "env": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "ASPNETCORE_URLS": "http://+:5002",
        «DAPR_HTTP_PORT»: «5020»
      },
      "sourceFileMap": {
```

```
        "/Views": "${workspaceFolder}/Views"
      },
      "postDebugTask": "daprd-down-webapi2"
    }
  ]
  ... omitted ...
```

As you can see from the launch configurations, each one refers to a corresponding `preLaunchTask` task and a `postDebugTask` task.

The `DAPR_HTTP_PORT` and `ASPNETCORE_URLS` environment variables match the configuration in the `task.json` file.

## Tasks

Here is an extract from the `tasks : [...]` element in the `task.json` file. You can find the complete configuration file in the sample directory:

```
... omitted ...
{
  "label": "build-webapi",
  "command": "dotnet",
  "type": "process",
  "args": [
    "build",
    "${workspaceFolder}/sample
      .microservice.webapi
      /sample.microservice.webapi.csproj",
    "/property:GenerateFullPaths=true",
    "/consoleloggerparameters:NoSummary"
  ],
  "problemMatcher": "$msCompile"
},
{
  "appId": "hello-world",
  "appPort": 5001,
```

```
        "httpPort": 5010,  
        "grpcPort": 50010,  
        "metricsPort": 9091,  
        "label": "daprd-debug-webapi",  
        "type": "daprd",  
        ... omitted ...  
    },  
    {  
        "appId": "hello-world",  
        "label": "daprd-down-webapi",  
        "type": "daprd-down"  
    }  
    ... omitted ...
```

In the previous snippet, you can see that for the `hello-world` service, corresponding to the `sample.microservice.webapi` project, the configuration adopts a set of `"appPort": 5001`, `"httpPort": 5010`, `"grpcPort": 50010`, and `"metricsPort": 9091`. The latter metrics endpoint is useful if you want to test Prometheus in your local development machine, as documented at <https://docs.dapr.io/operations/monitoring/metrics/prometheus/#setup-prometheus-locally>. In *Chapter 11, Tracing Dapr Applications*, we will learn more about this topic.

By contrast, the `salute-world` service configuration adopts other non-conflicting ports for the `appPort`, `httpPort`, `grpcPort`, and `metricsPort` settings.

It is worth noting that in the configuration file, two sets of tasks for `build`, `dapr-debug` (which launches the Dapr app), and `dapr-down` (to stop and clean up) have been created, one for each project.

I also had to edit the path to each project's folder with the `"${workspaceFolder}/<project path>/<file>"` pattern wherever a reference to the project's file or library was needed.

## Launching debug sessions individually

With these configurations, we can individually launch the integrated .NET and Dapr debugging experience in VS Code for each project. In the following screenshot, you can see the two debug configurations being recognized in VS Code:

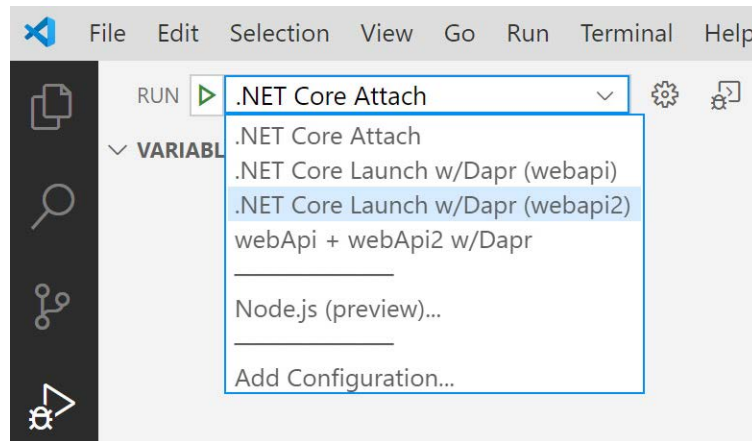


Figure 2.3 – Debug launch configurations in VS Code

We can test that each project is properly built and gets exposed (locally) as an ASP.NET endpoint, and then a Dapr CLI debugger is launched for each application.

As a simple test, in the following snippet, let's invoke the two Dapr applications via `curl` while they are in debugging mode in VS Code:

```
PS C:\> curl http://localhost:5010/v1.0/invoke/hello-world/
method/hello
Hello, World
PS C:\> curl http://localhost:5020/v1.0/invoke/salute-world/
method/salute
I salute you, my dear World.
```

Our first two sample Dapr services can be invoked via the Dapr runtime. Now let's try to launch a compound debug session next.

## Launching compound debug sessions

We are not done yet: our last objective is to instruct VS Code to build and launch the two Dapr applications at the same time, instead of having the user individually launch the configuration for each project. This can be achieved with a compound task in VS Code. You can visit [https://code.visualstudio.com/docs/editor/debugging#\\_compound-launch-configurations](https://code.visualstudio.com/docs/editor/debugging#_compound-launch-configurations) for more information.

A compound launch configuration starts multiple debug sessions in parallel. All we need to do is refer to the previously defined configurations in `launch.json`, as follows:

```
"compounds":
[
  {
    "name": "webApi + webApi2 w/Dapr",
    "configurations": [".NET Core Launch w/Dapr (webapi)", ".NET Core Launch w/Dapr (webapi2)"]
  }
]
```

As we can see from the following screenshot, we have both of the VS Code debug sessions active, and both of the Dapr applications we previously defined, `hello-world` and `salute-world`, are active:

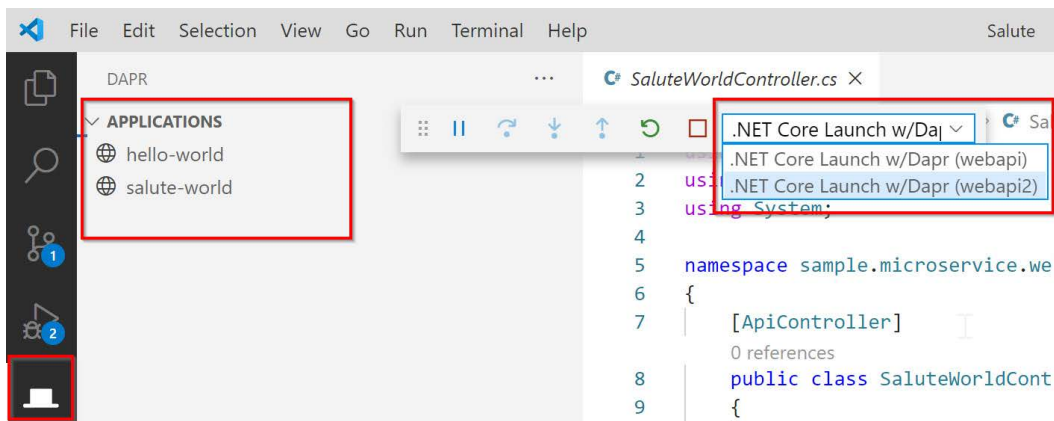


Figure 2.4 – A Dapr debug session in VS Code

As the Dapr applications have been launched from the Dapr CLI debugger, they appear in the context of the Dapr extension for VS Code, as shown in *Figure 2.4*, but they do not show up in the Dapr CLI; the `dapr list` command and the Dapr dashboard will show no applications running.

This completes our mission: we are now able to quickly enter a debug session that encompasses all the projects for the microservices and other libraries we need.

In the next section, we will learn about a new, exciting project in the .NET space that can help us improve the debugging experience with Dapr.

## Using Tye with Dapr

I find the multi-target debugging provided by VS Code a rewarding experience for Dapr solutions with several projects; nevertheless, there are always many options to accomplish the same task. One option for this particular task is **Tye**.

A new open source initiative by the .NET Foundation, Project Tye, is currently under development as an experimental project. For more information, check out the documentation at <https://github.com/dotnet/tye/blob/master/docs/>.

Project Tye is a tool to make developing, testing, and deploying microservices and distributed applications easier. In the upcoming sections, we will explore Project Tye in conjunction with Dapr.

## Installing Tye

At the time of writing, 0.10.0 is the latest public version of Tye. With the following command, you can install it in your environment:

```
PS C:\Repos\practical-dapr\chapter02> dotnet tool install -g
  Microsoft.Tye --version "0.10.0-alpha.21420.1"
You can invoke the tool using the following command: tye
Tool 'microsoft.tye' (version '0.10.0-alpha.21420.1') was
  successfully installed.
```

It's that simple! Tye is now ready to be configured in our .NET solution.

## Using Tye

In the root folder of the project structure, we initialize the `tye.yaml` file to instruct Tye on how to deal with the ASP.NET projects and Dapr configurations, as follows:

```
PS C:\Repos\practical-dapr\chapter02> tye init
Created 'C:\Repos\practical-dapr\chapter02\tye.yaml'.
Time Elapsed: 00:00:00:34
```

The preceding command creates a `tye.yaml` file with the following content:

```
name: hello-world-debug
services:
- name: dapr-microservice-webapi
  project: sample.microservice.webapi/sample.microservice.
  webapi.csproj
```

```
- name: dapr-microservice-webapi2
  project: sample.microservice.webapi2/sample.microservice.
  webapi2.csproj
```

As I already had a solution file, Tye recognized the presence of the two projects. In our previous samples, we used two different Dapr app-id instances: `hello-world` and `salute-world`; we will fix this in the next step.

By mutating the Dapr recipe from the Tye repository at <https://github.com/dotnet/tye/blob/main/docs/recipes/dapr.md>, I changed the default configuration to the following:

```
name: hello-world-debug
extensions:
- name: dapr
```

The only change has been adding extensions with `name: dapr`.

A caveat: as I already initialized Dapr on my local environment, Redis is already in place. Therefore, I left it out of the Tye configuration by commenting it.

From the command line (Windows Terminal or a VS Code terminal window), we can launch Tye with the `tye run` command, as follows:

```
PS C:\Repos\practical-dapr\chapter02> tye run
```

From the following output, we can see how Tye launches our Dapr applications:

```
Loading Application Details...
Launching Tye Host...
[21:23:05 INF] Executing application from C:\Repos\practical-
dapr\chapter02\tye.yaml
[21:23:05 INF] Dashboard running on http://127.0.0.1:8000
[21:23:06 INF] Building projects
[21:23:07 INF] Launching service hello-world-dapr_dclac8cb-5:
daprd -app-id hello-world -app-port 60146 -dapr-grpc-port 60150
--dapr-http-port 60151 --metrics-port 60152 --placement-address
localhost:50005 -log-level debug
[21:23:07 INF] Launching service hello-world_613eee7d-f: C:\
Repos\practical-dapr\chapter02\sample.microservice.webapi\bin\
Debug\net5.0\sample.microservice.webapi.exe
```

```
[21:23:07 INF] Launching service salute-world-dapr_90068b8d-4:
daprd -app-id salute-world -app-port 60148 -dapr-grpc-port
60153 --dapr-http-port 60154 --metrics-port 60155 --placement-
address localhost:50005 -log-level debug
... omitted ...
```

From the command output, we can observe that Tye launches the Dapr CLI debugger (daprd), referring to the corresponding Dapr application for the ASP.NET project, all by providing dynamic ports.

The Tye tool offers a portal from which you can access useful .NET metrics about the running processes, as illustrated in the following screenshot:

Name	Type	Source	Bindings	Replicas	Restarts	Logs
<a href="#">hello-world</a>	Project	C:\Repos\dapr-samples\chapter02\hello-world-debug\dapr.microservice.webapi\dapr.microservice.webapi.csproj	<a href="http://localhost:60146">http://localhost:60146</a> <a href="https://localhost:60147">https://localhost:60147</a>	1/1	0	<a href="#">View</a>
<a href="#">salute-world</a>	Project	C:\Repos\dapr-samples\chapter02\hello-world-debug\dapr.microservice.webapi2\dapr.microservice.webapi2.csproj	<a href="http://localhost:60148">http://localhost:60148</a> <a href="https://localhost:60149">https://localhost:60149</a>	1/1	0	<a href="#">View</a>
<a href="#">hello-world-dapr</a>	Executable		<a href="https://localhost:60150">https://localhost:60150</a> <a href="http://localhost:60151">http://localhost:60151</a> <a href="http://localhost:60152">http://localhost:60152</a>	1/1	0	<a href="#">View</a>
<a href="#">salute-world-dapr</a>	Executable		<a href="https://localhost:60153">https://localhost:60153</a> <a href="http://localhost:60154">http://localhost:60154</a> <a href="http://localhost:60155">http://localhost:60155</a>	1/1	0	<a href="#">View</a>

Figure 2.5 – The Tye portal

In *Figure 2.5*, you can see the Tye portal showing our Dapr services and the corresponding Dapr sidecars.

The `--debug *` option of the Tye command waits for VS Code debug sessions to be attached to each service: in the Tye portal logs, you will see a `Waiting for debugger to attach...` message.

In VS Code, we can leverage the `.NET Core Attach` debug configuration, by selecting the corresponding ASP.NET process (as we did at the beginning of this chapter) for each of our sample projects.

You might consider the option to customize the `.NET Core Attach` VS Code configuration to provide support for multiple projects, similar to what we did previously with the `.NET Core Launch w/Dapr (webapi)` debug configuration.

Once the VS Code debug session has started, we can invoke our Dapr applications, as follows:

```
PS C:\> curl http://localhost:60151/v1.0/invoke/hello-world/
method/hello
Hello, World
```

The first Dapr service responded; next, we test the second one, as follows:

```
PS C:\> curl http://localhost:60154/v1.0/invoke/salute-world/  
method/salute  
I salute you, my dear World.
```

The Tye tool is useful in the context of both debugging .NET solutions and deploying them. Its integration with Dapr greatly simplifies the debugging experience.

With Tye, we have completed our discussion of Dapr's debugging options.

## Summary

In this chapter, you have learned how to take advantage of several available options to debug C# Dapr projects, leveraging VS Code's debug capabilities and configurations, the Dapr CLI itself, and Project Tye.

Getting comfortable with Tye and the Dapr CLI is also important, as these will be the most frequently used approaches to launch our Dapr application in the testing phase.

In the next chapter, we will explore in more depth how services are invoked within Dapr.

## Questions

1. How do you scaffold debugging for Dapr in VS Code?
2. How do you compound the debug configuration for multiple Dapr projects?
3. How do you attach to a running Dapr application with VS Code?
4. How do you configure the Dapr components in Tye?

## Further reading

- Debugging Dapr with VS Code: <https://docs.dapr.io/developing-applications/ides/vscode/vscode-how-to-debug-multiple-dapr-apps/>
- The VS Code extension for Dapr: <https://docs.dapr.io/developing-applications/ides/vscode/vscode-dapr-extension/>
- Tye for Dapr: <https://github.com/dotnet/tye/blob/main/docs/recipes/>

# Microservices Architecture with Dapr

In this chapter, I intend to discuss the relevance of microservices architectures for building modern applications, as well as exploring how Dapr, as a runtime, can make this cultural shift way easier to adopt to achieve the many advantages that come with it. Learning why and how microservices are beneficial will help you make better use of Dapr, too.

In this chapter, we're going to cover the following main topics:

- Introducing our sample solution
- Discovering microservices
- Adopting microservices patterns
- Building an e-commerce architecture
- Building microservices with Dapr

We'll start by introducing the sample solution we will build over the course of this book.

## Introducing our sample solution

In order to navigate the concepts of Dapr throughout this book, we could use the help of a common theme to guide us. I thought that a fictional scenario, with an architecture that we will implement together, might be a good approach for the samples.

Therefore, let's introduce the architecture of *Biscotti Brutti Ma Buoni*. In our quest to explore Dapr, it's time to move away from the Hello-World kind of samples and shift to a hypothetical e-commerce scenario, which will be presented from different perspectives to illustrate each of Dapr's capabilities over the course of the following chapters.

The architecture of the fictional e-commerce site *Biscotti Brutti Ma Buoni* (which means *ugly but good cookies* in Italian) is further discussed from a microservices perspective in *Appendix, Microservices Architecture with Dapr*.

In a nutshell, the *Biscotti Brutti Ma Buoni* website sells cookies to consumers, offering the ability to customize them. Cookies are manufactured all day; therefore, keeping track of their availability is important. Hopefully, this scenario will help us learn new topics from a practical standpoint and will make it easier to understand and implement them.

Let's start by exploring the basic concepts of designing microservice architecture.

## Discovering microservices

There is an endless collection of books, papers, and blog posts that excellently describe and analyze microservice-style architecture. The objective of this chapter is to present you with the advantages and challenges of using a microservices architecture and to find out how Dapr can help us create new applications based on it.

The nemesis of a microservices architecture is the monolith. No one would ever admit they built or are still working on one, but most of us in the development industry have spent many years working on monolith applications. In a monolith, as the name implies, all the business capabilities or features are condensed in a single application, probably layered between the UI, server, and database, but nevertheless not designed in a modular or distributed fashion.

In a microservice architecture, the services that are designed to support business capabilities are most likely to communicate with open protocols such as HTTP and gRPC. Microservices are built and released via automation, can be independently deployed, and each microservice's team can adopt the language and technology stack that best fits their needs.

As microservices are an evolution of the **Service-Oriented Architecture (SOA)**, let me briefly summarize the overall architectural concepts we are dealing with in the following sections:

- Service
- Autonomy
- Automated deployment
- Bounded context
- Loose coupling
- Event-driven architecture
- Observability
- Sustainability

We will look at these individually in the following sections. Let's start by exploring the core concept of a service.

## Service

A service is a logical representation of a process that's capable of fulfilling a business activity: selling a product to a potential customer, supporting a customer in fixing an issue with the product, and so on.

A service exposes an **application programming interface (API)** that regulates how interaction with the service can occur.

It is very likely that the API will be defined as a REST API over HTTP, as most of the developer audience would expect in today's world, but this concept is not restricted to a specific implementation detail. As an example, gRPC is also a popular choice for exposing a service.

From the client's perspective, there is no interest in being, or needing to be, aware of the internal mechanisms of the service. What matters to the client is the API contract, its stability over time, and the **service-level agreement (SLA)**, which indicates the objectives of availability and performance the service promises to uphold, as well as the reparations, in case it can't keep those promises.

## Autonomy

A microservice should be autonomous in reaching its objective, as declared via the API, regardless of its dependencies toward other services. Autonomy applies to operations and evolution as well.

To gain more autonomy, the service should limit the need to coordinate with dependent services: the more dependencies a service has, the more difficult it will be to respect the desired availability and to evolve, as changes to other services will impact your service over time.

Let's look at an example: your application has two microservices, each relying on a common portion of code. As you identify this common ground, a case for a new microservice is made by separating the common portion of code into a new service. Does this make any sense? Not from my perspective: instead, consider refactoring the common code as a library and distributing it via NuGet, the package manager, so that the two microservices are free to control which version they use and when to update it.

Is this common portion of code the starting point of a new business capability that will grow over time? This would shed a different light on the scenario, which is more favorable for a third microservice that will introduce more coordination.

There should be a very good reason to add coordination to a dependent service, since having others depend on your microservice is a responsibility not to be taken lightly.

## Automated deployment

By introducing microservices, you will tend to have more independently deployable units of code (which I see as a benefit) for several reasons. For example, different teams may be working on separate projects. Whatever the reason, you will need to adopt continuous deployment practices and tools to perform these operations in an automated fashion. Otherwise, if performed manually, the build and deployment process of microservices could be much more time-consuming than those of a monolith application.

## Bounded context

A bounded context is a pattern that originated in the domain-driven design space. The unified model of an enterprise/large application tends to grow significantly large, so it becomes intrinsically difficult to manage over time. Splitting the complex model into smaller, more independent integrated models, each with a coherent purpose, can be a solution.

As an example, an e-commerce application might be difficult to manage as a single model. Can we agree that separating the context of sales from after-sales (support, products return, complaints handling, and so on) might be a more simplified approach?

The *micro* in microservices suggests the idea of *small*, but according to which measure? Is it the number of classes? The portable executable size as a file? The overall number of libraries?

In my opinion, the word *micro* does not help clarify the concept, while a bounded context is far more useful to indicate that a microservice should take care of a single part of the application. The reduced size of a microservice compared to a monolith is often a byproduct of this.

## Loose coupling

Directly from the SOA space, the interactions between two services should be as loosely coupled as possible: if you reach this goal, you could deploy a service without impacting others.

This goal can be reached by implementing patterns that favor an indirect interaction (publish-subscribe) over a direct request-reply. While it's not easy to keep services loosely coupled, this might prove an insurmountable task, so further analysis should verify whether the services (or microservices) in scope are meant to be kept separate or should be combined instead.

If this seems similar to the concept of autonomy, you are right. As a matter of fact, the looser you couple two microservices, the more autonomous they are.

## Event-driven architecture

Event-driven is an architectural pattern in which events, such as changes in the state or facts, are produced for others, unknown to the producer, to be noticed and eventually consumed. For example, a new sale and the quantity of a product reaching 0 (which means it is out of stock) are two types of events.

With this pattern, a service is triggered not by a direct invocation, but by a message representing the event being detected. Inherently, both parties – the producer and the consumer – operate at a scale and speed that fits their own objectives, not interfering with each other.

To build an event-driven architecture, you will likely have to leverage a message bus to handle the complexity of exchanging messages.

## Observability

A microservice architecture brings many more moving parts into play since they're deployed more frequently, probably in a greater number of instances, over many hosts. Therefore, it is paramount to gain full visibility of the status and behavior of each microservice instance, each point of coordination, and each component of the underlying platform/runtime, making it easy for both the operators' and developers' audiences to gain easily readable and actionable intelligence from this vast group of information.

In contrast to our usual experiences, counting on accessing the logs from all the nodes will not help much; having a synthetic probe result letting you know the microservice is up is useful, but it won't tell you that the microservice is demonstrating its ability to consistently perform its business capability over time. What you want to achieve is to have full traceability for each request as it crosses all the boundaries between microservices and the underlying stores and services.

## Sustainability

Given the current historical context, where we are facing a climate crisis, there is much interest in sustainable operations, and lately, I have witnessed the growth of a community for sustainable software engineering.

**Sustainable software engineering** is a new discipline that combines climate science with software architecture – the way electricity is generated with how data centers and hardware are designed. You can learn more about this at <https://principles.green>.

The overall picture is that developers, software analysts, and even application owners should consider the carbon impact of their software, and all the possible ways to reduce it or at least adapt it to the conditions of the underlying energy offering. A perfect example of this discipline is the low carbon Kubernetes scheduler (white paper available at [http://eur-ws.org/Vol-2382/ICT4S2019\\_paper\\_28.pdf](http://eur-ws.org/Vol-2382/ICT4S2019_paper_28.pdf)), which helps developers move their application's containers according to the lowest carbon impact of the current location. I think Dapr can also influence this movement. This is because it makes microservice applications much easier to write and handle – adding the sustainability option should not be a huge effort – and besides, architecting a piece of software to not waste resources should be in our best interests anyway.

Since we share a common understanding of the core concepts of a microservice architecture, let's look at the benefits its adoption provides.

## Adopting microservices patterns

What benefits does a microservice architecture bring to an application? Why should we move from monoliths to microservices?

Considering the concepts described so far, I think the following might be a good list of improvements you can achieve by adopting microservice architecture:

- **Evolution:** It is common to have several smaller teams, each working with the tools, languages, and platforms that best match their objectives. By defining a simpler and smaller (bounded) context of the application, there are far better chances that its evolution and growth will be faster, more reliable, and have greater business impact.
- **Flexibility:** By becoming as autonomous as possible, and by interacting with others in a loosely coupled manner, the microservice will gain many opportunities that would otherwise be impossible for a monolith architecture: changing the persistence layer, adopting a new library, or even switching to a different technology stack now becomes a possibility. As long as there is no impact on other microservices, each team is free to choose their own innovation path.
- **Reliability:** Code quality will not improve by itself just because it is in a microservice but having a smaller scope and being able to deploy it individually makes it easier to test it in an automated fashion, and it will have increased reliability as a byproduct.
- **Scale:** Microservices enable you to deploy as many instances as you need, dynamically increasing and reducing them (scale out and in), and even independently choosing the right resource, such as a specific database or a virtual machine type (scale up and down), equipped with a specific CPU your code can leverage. These newly acquired abilities enable your application to achieve a much higher throughput rate, more efficiently than with other approaches.

### Important note

With merit to scale, microservices are often cited as the default architecture for applications landing on Kubernetes because of their ability to scale and coordinate resources.

Kubernetes is a good choice for hosting a microservice architecture, as is a **Function as a Service (FaaS)** offering such as Azure Functions, to some degree.

In FaaS, the focus is only on the application code, with a greater abstraction from the infrastructure layer than there is for a deployment based on containers.

The main thing to note is that you should keep the subjects separate; Kubernetes can sometimes be the proper hosting platform for monolith applications that have been condensed into a single Pod.

All the benefits a microservice architecture brings to the application translate into advantages for your product against the competition.

This book only focuses on .NET, which we all deeply love, but it will probably not be the only language that's used by your application. To support this common scenario, Dapr provides HTTP and gRPC APIs, and SDKs for most diffused languages. If your application needs to leverage machine learning, whether for existing models or to train new ones, these are disciplines in which the audience is more comfortable with Python, just to start with the obvious. Therefore, being able to fully leverage the job market, because your architecture paradigm allows you to choose the best technology for the task, can be a strategic advantage that you should not underestimate.

It is a good exercise to ask ourselves the following question: *should we always adopt microservices?*

Microservice architecture is the distilled outcome of endless improvements in the way software is designed and built, so it's up to us, architects and developers, to adopt the latest and best in our daily jobs.

Nevertheless, I think every technology should only be adopted if it brings a net benefit.

If there is only a team (containing a few members) working on the application, splitting a monolith into microservices will not make it easier for that team to cope with the burden of more projects, more complex dependencies, and separate CD pipelines, just to name a few.

Regarding infinite resources, a monolith application may be able to scale to meet your goal by adding new instances of your host of choice, such as a virtual machine. As long as more powerful virtual machines become available, your monolith application will always have room for growth.

The scale for which microservices emerged as a winning architecture may be way beyond your needs. On the other hand, monoliths were not built to deal with the complexity of managing updates and coordination over too many hosts and instances.

If the team does not reach proper maturity on automated testing and CI/CD (namely, the overall DevOps culture), it might be a significant challenge to adopt microservices and handle all the aforementioned issues at the same time. Nevertheless, if time and resources are in your favor, you can try to kill two birds with one stone.

Finally, it may become difficult to identify clear context boundaries in your applications, maybe because they fulfill a very specific task and not much more. In these cases, there may not be a clear benefit of splitting it further, needlessly, into microservices. Nevertheless, adopting the same patterns and aiming for the same goals of microservice architecture may help you in the long run.

Let's apply the concept discussed so far to our sample architecture.

## Building an e-commerce architecture

The objective of this book is to illustrate how Dapr can support your job, as developers and architects, in creating an application that adopts a microservice architecture. I think it has helped to discuss a hypothetical scenario over the course of this book to see how each feature can be introduced. I considered a scenario we will all experience once or more in our daily lives, mostly as consumers (sometimes as creators), when building an e-commerce site.

An e-commerce site must support many capabilities: exposing a catalog and making it browsable, having a price model that can be influenced by a promotion engine, managing a shopping cart, collecting customer information, processing orders, fulfilling the orders, and coordinating the shipments, just to name a few.

Throughout this book, we will compose a sample solution for the fictional e-commerce site named *Biscotti Brutti Ma Buoni*. This site specializes in baking and customizing cookies. Most of its customers don't buy the plain cookies packages but instead order customized versions for special occasions. *Biscotti Brutti Ma Buoni* is widely known for its ability to digitally print complex, multilayer scenes on a cookie.

Starting with this fictional example of e-commerce, we will explore the context of its components, all of which we want to include as microservices in our solution.

## Bounded contexts

As we couldn't interview the business and domain expert of the fictional e-commerce site, I briefly ask for your suspension of disbelief, as if you were watching a play. Let's agree that we identified several bounded contexts:

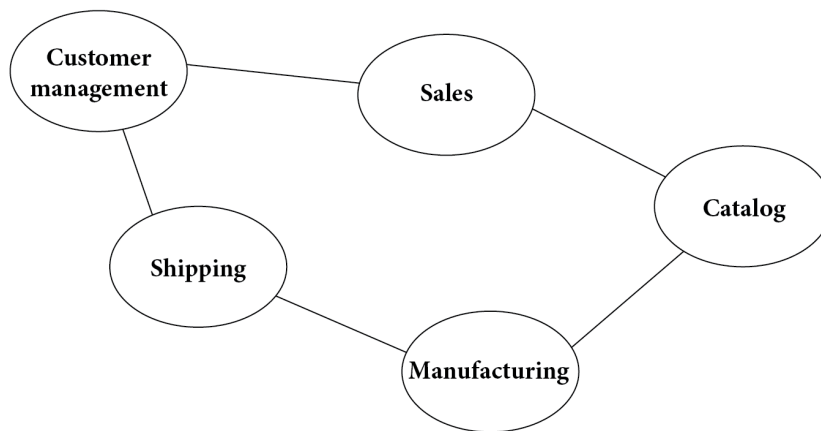


Figure 3.1 – The bounded contexts

Let's focus our attention on a few of the contexts shown in the preceding diagram:

- **Sales:** Most of the items sold are made on demand and may need to be customized. Therefore, sales orders only tend to be accepted if the manufacturing process is active. The stock is relevant during the manufacturing process. If for any reason a product line is temporarily unavailable because a particular ingredient is missing or a piece of equipment breaks, the order might be put on hold or canceled, and the item would become unavailable in the front store.

- **Manufacturing:** Luckily for *Biscotti Brutti Ma Buoni*, most of the items on sale are made from the same dough, so planning for the main ingredients is simple. Customization is more complex as colorants and other ingredients for icing are much more diverse. The request to manufacture the final item is guided by the actual sale, but the provisioning of ingredients is driven by a machine learning model that takes seasonal and short-term sales into account. Manufacturing also encompasses customizing the cookies.
- **Shipping:** While shipping is important to all e-commerce sites, as nowadays customers are used to immediate delivery, for perishable and delicate goods, order fulfillment demands maximum care.

To facilitate the exploration of Dapr, the connection between the bounded contexts will prefer coordination via service invocation and pub/sub instead of data duplication or any other approach.

### An example – sales microservices

As an example, let's suppose that analyzing the sales-bounded context further is going to require the following microservices:

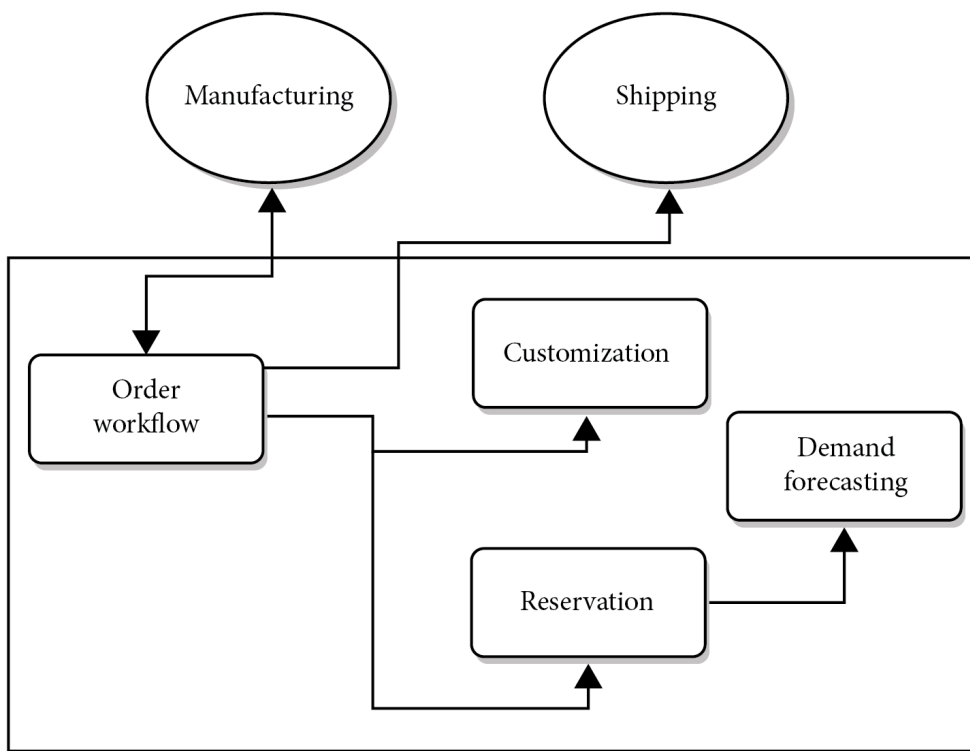


Figure 3.2 – The sales-bounded context

As we can see, a sales order triggers the preparation workflow via a loosely coupled interaction. The **Reservation** service will also be informed by consuming events on the order status update. Once the order preparation workflow successfully completes, it will trigger **Shipping** as the next step.

As we have identified a few suitable bounded contexts that we explored in our sample solution, let's find out how Dapr supports microservices architecture.

## Building microservices with Dapr

How can Dapr help us build this e-commerce application by adopting a microservice architecture?

In this section, we will learn about the specific benefits Dapr brings to a microservice architecture. Let's start by exploring them all and begin with loosely coupled microservices.

### Loosely coupled microservices

With pub/sub in Dapr, we can achieve two objectives. Not only does Dapr make it transparent to use any of the supported messaging systems, such as Redis, RabbitMQ, Azure Service Bus, and Azure Event Hubs, but it also provides all the plumbing code that's responsible for handling the message operations, ensuring at-least-once delivery.

Two microservices, signaling to one another an event via pub/sub, can coordinate via a loosely coupled connection. If the consumer is experiencing a temporary issue, the information that's sent by the producer will stay safely in the messaging subsystem of choice, waiting for the consumer to come back and get it.

### Autonomous microservices

A service with Dapr can be invoked by just specifying an application identifier. It is then Dapr's responsibility to discover where the service is running in the hosted environment (most likely Kubernetes), how to reach it, and how to handle its request/response via a secure communication channel.

If the invoked service, over time, evolves to adopt a different set of libraries in order to change the data layer or the data storage itself, it will appear and operate just the same.

### Observable microservices

The microservices that were identified in the manufacturing bounded context interact with each other and with other bounded contexts as well, and they communicate between many different hosting environments. This includes the Nodes and Pods of Kubernetes and their state stores, the messaging systems, and more.

It soon becomes clear that while a collection of logs from the Kubernetes infrastructure is helpful, what is much more needed is distributed tracing for each activity as its processing flows from one step to the next, from one microservice to the other, by sharing a common context of the customer journey.

## Scalable microservices

Dapr promotes the usage of Kubernetes as the hosting platform for your microservices, enabling dynamic and rapid scaling of each independently on as many resources, Pods, and Nodes as needed.

Over the course of this book, we've learned how easy it is to create microservices with Dapr. From this perspective, Dapr enables architects and developers to only consider the bounded context and microservice analysis when it comes to defining the implementation details of the architecture. The unnecessary proliferation of microservices is to be avoided, and Dapr will not push your architecture toward more microservices or less, but it will significantly lift the initial effort needed.

## Event-driven microservices

An event-driven architecture can be achieved in many ways: as an example, I can have a loop in my code that monitors the messaging or external subsystems, via a long polling approach, for new events.

In this scenario, I would be responsible for keeping the process active, regardless of whether I am relying on a PaaS or IaaS hosting environment. My code could leverage a library to spare me from the inner details of the message system, but nevertheless, I am still influenced by the process and host recycling as I must keep listening for events. Only at the end of this complex chain of elements will I have the code expressing my business logic.

There is a subtle but important difference: not having a library but counting on a runtime, designed to operate in complex conditions, in an environment such as Kubernetes, capable of fast recovery, reduces the code under your responsibility to just the message handling logic and is a tremendous advantage offered by Dapr.

## Stateless microservices

A stateless microservice is easier to distribute over as many instances as needed, has a faster life cycle, and is more solid when it comes to handling faults and conditions of error.

Nevertheless, many – if not most – of the microservices we create need to manage a state, whether it is used to support how a request is processed or it represents the core data the code is handling.

By managing state as a service, with pluggable interchangeable components, Dapr makes any microservice practically a stateless microservice. With the reliable state API provided by Dapr, the complexity of managing state by taking concurrency and consistency into account is lifted from the service code.

This section helped us understand in which ways Dapr can help us in developing microservices architecture, knowledge that will guide us in our future endeavor in developing the architecture for the *Biscotti Brutti Ma Buoni* solution.

## Summary

In conclusion, the reason why Dapr has an immediate appeal to native cloud developers is its ability to provide flexibility and simplicity in a very complex environment. As a fellow Microsoft colleague often said, *“Only 10 years ago, an entire developer’s career could be built on what today is just a simple scaling slider.”*

This is the future of development, and in addition to the native cloud tools, Dapr is also offering a chance to bring together all the possible combinations of legacy applications and programming languages, so that a complete refactoring is no longer the only modernization option.

After this overview of microservice architecture, in the next chapter we will explore Dapr in more depth, starting with how services are invoked within Dapr.

## Questions

1. Why are bounded contexts so important?
2. Should I implement a microservices architecture when designing a new solution?
3. What is the most relevant contribution by Dapr to microservices?

## Further reading

- Bounded context, by Martin Fowler: <https://martinfowler.com/bliki/BoundedContext.html>
- Microservices, by Martin Fowler: <https://www.martinfowler.com/articles/microservices.html>
- Microservices architecture, by Mayro Servienti: <https://particular.net/videos/microservices-architecture-is-it-the-right-choice-to-design-long-living-systems>
- Microservices architecture: <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/microservices-architecture>

# Part 2: Building Microservices with Dapr

With our newly acquired knowledge of the basics of Dapr, we'll learn why microservices architecture is important and how Dapr can help you on the journey to implementing it.

This part contains the following chapters:

- *Chapter 4, Service-to-Service Invocation*
- *Chapter 5, Introducing State Management*
- *Chapter 6, Publish and Subscribe*
- *Chapter 7, Resource Bindings*
- *Chapter 8, Using Actors*



# Service-to-Service Invocation

In this chapter, we will learn how **Distributed Application Runtime (Dapr)** applications can communicate with each other via the **Dapr** infrastructure.

We are going to understand this with the following main topics:

- Invoking services with Dapr
- Service invocation with the .NET SDK
- Comparing HTTP and gRPC to Dapr

With hands-on examples, we will understand how to implement services and invoke them from other applications, which can be either aware of the presence of Dapr, as they rely on its SDK, or unaware, as they just invoke a local HTTP endpoint.

Before we start using service-to-service invocation, a building block of Dapr, let's understand how it works, using an example.

## Technical requirements

The code for this chapter can be found on GitHub at <https://github.com/PacktPublishing/Practical-Microservices-with-Dapr-and-.NET-Second-Edition/tree/main/chapter04>.

In this chapter, the working area for scripts and code is expected to be `<repository path>\chapter04\`. In my local environment, it is `C:\Repos\practical-dapr\chapter04`.

Please refer to the *Setting up Dapr* section in *Chapter 1, Introducing Dapr*, for a complete guide on the tools needed to develop with Dapr and work with the samples.

## Invoking services with Dapr

In this section, we will learn how Dapr provides our microservices with the ability to interact directly via service-to-service invocation.

**Services** are the centerpiece of Dapr. A service in Dapr enables a developer to easily make the API of a microservice discoverable and reachable to other components inside the hosting environment, whether it be a self-hosted or Kubernetes cluster.

The Dapr service invocation API, which we will leverage via the abstraction offered by the Dapr .NET SDK, provides discovery and reliable communication, with standard protocols such as HTTP and gRPC.

In previous chapters, we built a few Dapr service samples, but we must give proper attention to the details, which we will do in this chapter. How can a service be reached via Dapr? That is going to be the focus of this chapter:

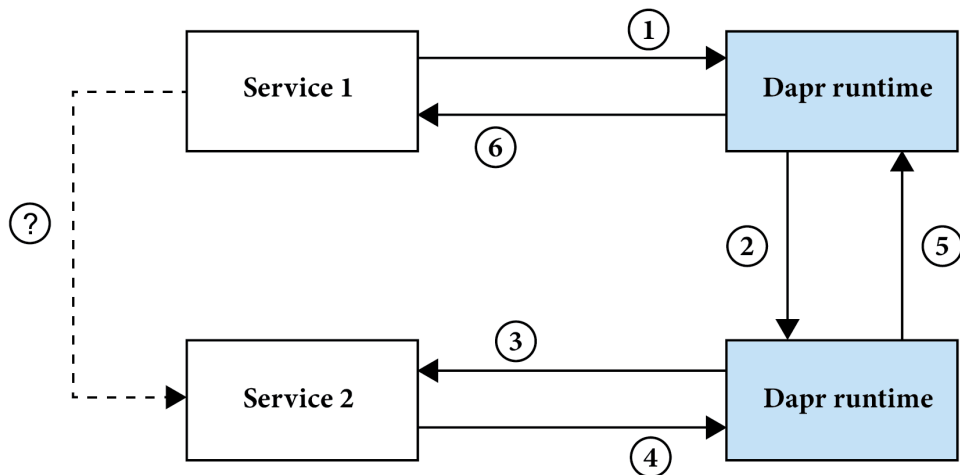


Figure 4.1 – Service-to-service invocation in Dapr

In *Figure 4.1*, we can visualize the path requests and responses taken to reach a specific service, by traversing the **Dapr runtime**, which runs in a sidecar. In our local development environment, this is a simple interaction between local processes, while on Kubernetes, it will be a more complex interaction between sidecar containers automatically injected by Dapr into each of our application's Pods.

This is what happens during service-to-service invocation, as shown in *Figure 4.1*:

1. Once **Service 1** makes a call intended to reach **Service 2**, the call from **Service 1** reaches the Dapr runtime on the local sidecar, which discovers the location for **Service 2**.
2. The runtime on the local sidecar forwards the request to the Dapr local sidecar of **Service 2**.
3. This Dapr sidecar invokes **Service 2** on the configured application port.
4. The same sidecar receives back the result from the application logic.
5. The result is, in turn, returned by the **Service 2** Dapr's sidecar to the **Service 1** Dapr's sidecar.
6. At last, the result of the request to **Service 2** is returned to **Service 1**.

We will now learn how to implement Dapr services in ASP.NET in the following sections.

## Introducing service-to-service invocation

As the first sample in our fictional scenario, let's consider an ordering system whose API receives a fully composed order with items and quantities. Its first objective is to allocate the requested items, by reserving the quantities if it is a new order or adjusting the quantities if the order has been updated after the initial submission. We can depict two microservices: **order** and **reservation**:

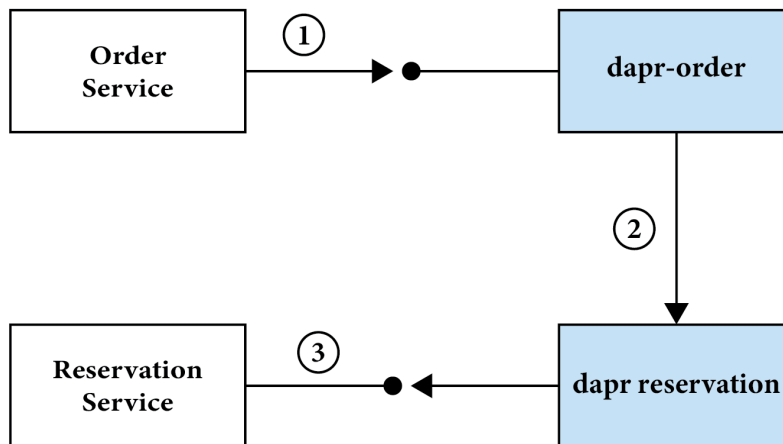


Figure 4.2 – The order and reservation services

From clients outside of this partial perspective, the **order** service receives a new or updated order. It needs to interact with **reservation**, another Dapr service, as follows:

1. From the **order** service, which is an ASP.NET Web API, a number of calls to the Dapr local endpoint are made in the following form:

```
POST http://localhost:3500/v1.0/invoke/reservation/
method/reserveitem
```

The methods supported by the Dapr service are POST, GET, DELETE, and PUT. In this example, we could choose to react to POST for new orders and PUT for updated orders. The default Dapr sidecar port is 3500 inside the Pod of a Kubernetes environment, while in our local development, each sidecar has to use a different port number, and **reservation** is the name of the Dapr service we want to reach to invoke the **reserveitem** method.

Communication between the service and the local sidecar can happen over HTTP or gRPC.

2. The local Dapr sidecar identifies the Dapr sidecar capable of reaching the **reservation** service and forwards the request.

This leg of communication between Dapr sidecars, outside of the developer's control, happens via gRPC and is secured via mutual TLS.

In a Kubernetes hosting environment, chances are it will traverse different nodes over the network.

3. The Dapr sidecar for the **reservation** service has the knowledge to reach the service on the local port to request the **reserveitem** route. As an ASP.NET API, it is likely to be exposed on port 80 via Kestrel as a container in a Kubernetes Pod, while in local development, we have to use different ports to avoid conflicts. The call would have this form:

```
POST http://localhost:80/reserveitem
```

The **reservation** service can be exposed as HTTP or gRPC.

Let's recap how Dapr handles URLs in the service-to-service invocation building block, by examining the following request:

```
http://localhost:3500/v1.0/invoke/<service>/method/<path>
```

By default, the Dapr sidecar can be reached at port 3500, which can be changed via configuration, at localhost. With **invoke**, the code is selecting the building block to reach the Dapr application named <service>. Everything after **method** will be the path passed by the corresponding Dapr sidecar to the destination service.

Let's take the following example:

```
http://localhost:3500/v1.0/invoke/appA/method/do/this/that
```

We can assume that the Dapr sidecar on the receiving end will pass the whole path to our appA application, as follows:

```
http://localhost:80/do/this/that
```

Dapr does not interfere with the path, payload, or headers that a service passes to another service.

After covering the basics of how a service can be reached via Dapr compared to directly via the ASP.NET endpoint, let's find out how name resolution works.

## Name resolution

**Service discovery** is the component that enables any request to a Dapr sidecar (process or container) to identify the corresponding sidecar and reach the intended service endpoint.

Dapr relies on Kubernetes name resolution in this hosting mode and on **Multicast DNS (mDNS)** while in self-hosted mode, such as in a local development environment.

Before we start exploring the .NET SDK, let's learn about resiliency in Dapr.

## Resiliency

As we learned in *Chapter 3, Microservices Architecture with Dapr*, we should assume that in a microservices architecture, the instances are spread over multiple nodes, possibly requiring a network traversal for a microservice to reach another one. Therefore, an application's ability to properly deal with transient failure is very important.

Dapr provides a retry mechanism in the service-to-service building block by default to handle failure and transient errors. This mechanism can be further tweaked to accommodate our application needs with the resiliency feature, introduced as a preview feature in Dapr version 1.7. Let's see how this feature works.

With resiliency, we can configure policies and apply these to targets. A target can be a specific Dapr application, component, or actors. Policies can be of the following types:

- **Timeouts:** Simply specify the maximum duration of an operation, after which the operation is considered as failed.
- **Retries:** Retry the operation, waiting for a constant or exponential time interval between attempts, possibly limiting the maximum number of retries.

- **Circuit breakers:** Execute operations while a circuit is closed; when a certain error condition is evaluated to be true, the circuit is opened to prevent further failures. After a timeout, the circuit is half-opened to let a maximum number of operations pass: if everything is fine, the circuit is closed once again, allowing the normal flow of operations to reach the target.

The circuit breaker is a very interesting policy, as it helps to reduce pressure on an application while it is experiencing transient failures, leaving it enough time to restore a healthy status before resuming the usual pace of operations. You can read more about the pattern at the Microsoft Architecture Center: <https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker>.

Let's see how to apply resiliency to our scenario:

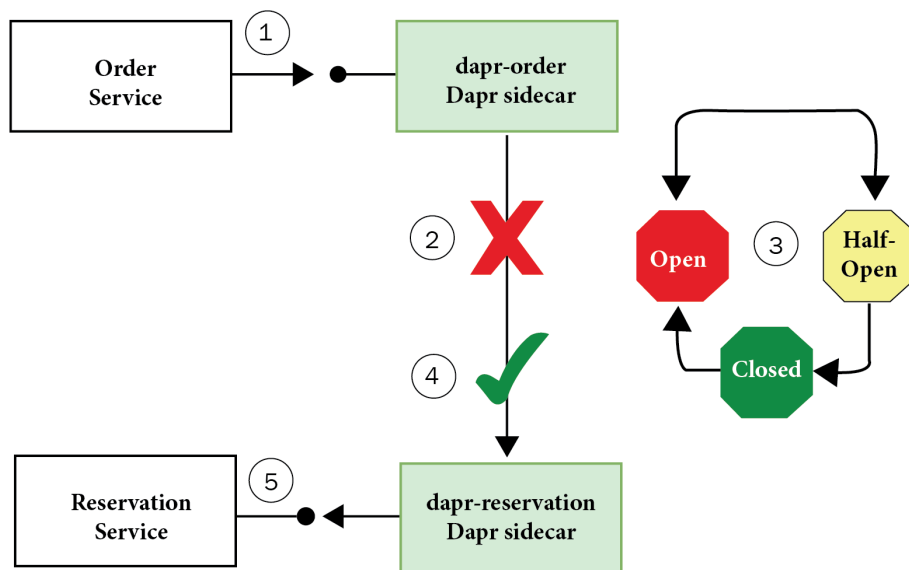


Figure 4.3 – The resiliency policy of the circuit breaker type

In *Figure 4.3*, we can see an example of the circuit breaker in action. While everything is fine, the circuit is open and operating. As soon as there are more than three consecutive failures over 5 seconds, the circuit is open (*step 2*) and no further requests from the **Order Service** can reach the **Reservation Service**. After a timeout of 30 seconds, the circuit (*step 3*) is half-opened to let one request pass; if successful, the circuit is closed again (*step 4*).

To leverage the resiliency feature, we need to configure policies and targets in a `.yaml` file. In *Chapter 5, Introducing State Management*, we will start to learn more about how to use `.yaml` files to configure Dapr; for the time being, to enable the aforementioned scenario, let's observe the content of the `components\resiliency.yaml` file:

```
apiVersion: dapr.io/v1alpha1
kind: Resiliency
metadata:
  name: myresiliency
spec:
  policies:
    ... omitted ...
  circuitBreakers:
    commonCB:
      maxRequests: 1
      interval: 5s
      timeout: 30s
      trip: consecutiveFailures > 3
  targets:
    apps:
      reservation-service:
        timeout: general
        retry: important
        circuitBreaker: commonCB
  ... omitted ...
```

The preceding configuration indicates, among other things, a circuit breaker to trip if there are more than three consecutive failures over 5 seconds, and the circuit will be half-opened after 30 seconds to let one more request pass. If no error arises, requests continue to flow without limitations.

With the `targets` part, we configure which circuit breaker, as well as which timeout and retry settings, to use for the Dapr `reservation-service` app.

In order to use the Resiliency feature in the self-hosted environment, the command must specify `component-path` and `configuration`, as shown in the following command snippet:

```
PS C:\Repos\practical-dapr\chapter04\sample.microservice.order>
dapr run --app-id "reservation-service" --app-port "5002"
--dapr-grpc-port "50020" --dapr-http-port "5020" --config
"./previewConfig.yaml" --components-path "./components" --
dotnet run --project ./sample.microservice.reservation/sample.
microservice.reservation.csproj --urls="http://+:5002"
```

In the following chapters, we will learn about other types of components and configurations. In *Chapter 9, Deploying to Kubernetes*, we will learn how to apply these components to a Dapr application running on a Kubernetes cluster.

As the Resiliency feature is in preview, it must be explicitly enabled; please see the Dapr documentation at <https://docs.dapr.io/operations/support/support-preview-features/> on how to configure it. The samples we will explore over the following chapters do not require the use of a Resiliency configuration; nevertheless, I strongly suggest you take this as an additional challenge and experiment with it.

The resiliency feature applies not only to service-to-service invocation but also to state management, input/output binding, and actor building blocks.

The next sections will be dedicated to .NET, as well as HTTP and gRPC service invocation methods.

## Service invocation with the .NET SDK

The best approach to understanding how the Dapr .NET SDK supports service-to-service invocation is to thoroughly examine a sample project.

These are the steps that we will follow:

1. Create a project for the **Order** service.
2. Configure Dapr in ASP.NET.
3. Implement Dapr with an ASP.NET controller.
4. Create a project for the **Reservation** service.
5. Prepare debugging configuration.
6. Implement Dapr with an ASP.NET Minimal API.

We will start by creating a project for the first Dapr application in our solution.

### Creating a project for the Order service

When creating a project for a Dapr service, we start from the `webapi` template:

```
PS C:\Repos\practical-dapr> cd .\chapter04\  
PS C:\Repos\practical-dapr\chapter04> dotnet new webapi -o  
sample.microservice.order  
The template "ASP.NET Core Web API" was created successfully.  
Processing post-creation actions...  
Running 'dotnet restore' on sample.microservice.order\sample.  
microservice.order.csproj...
```

```
Restore completed in 151,4 ms for C:\Repos\practical-dapr\chapter04\sample.microservice.order.csproj.  
Restore succeeded.
```

We have now created our project from the template. Next, we will configure ASP.NET.

## Configuring Dapr in ASP.NET

Dapr provides several libraries via NuGet (<https://www.nuget.org/profiles/dapr.io>) for .NET. The most relevant ones are as follows:

- `Dapr.AspNetCore`
- `Dapr.Client`

With the following command, we can add the packages to our .NET project:

```
PS C:\Repos\practical-dapr\chapter04\sample.microservice.order>  
dotnet add package Dapr.AspNetCore --version 1.6.0
```

We now have an ASP.NET project with full support for Dapr. Let's add some Dapr-specific code to our ASP.NET controller.

## Implementing Dapr with an ASP.NET controller

To configure Dapr in an ASP.NET project with .NET 6, we first have to follow the same instructions as in *Chapter 1, Introducing State Management*, in the *Building our first Dapr sample* section.

In the `Program.cs` file of our `sample.microservice.order` project, we are leveraging .NET 6 minimal hosting to set up Dapr integration:

```
var builder = WebApplication.CreateBuilder(args);  
var jsonOpt = new JsonSerializerOptions()  
{  
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,  
    PropertyNameCaseInsensitive = true,  
};  
builder.Services.AddControllers().AddDapr(opt => opt.  
UseJsonSerializationOptions(jsonOpt));  
... omitted ...  
var app = builder.Build();
```

```
... omitted ...
app.MapControllers();
app.MapSubscribeHandler();
app.Run();
```

The `AddDapr()` method integrates Dapr with the ASP.NET framework.

This chapter focuses on service-to-service invocation; therefore, the `MapSubscribeHandler()` method would not be necessary. However, it does enable the Dapr publish and subscribe building block via the SDK, as we will learn in *Chapter 6, Publish and Subscribe*. Nevertheless, I kept it for uniformity of the sample projects.

#### ASP.NET Naming

According to the announcement at <https://devblogs.microsoft.com/dotnet/announcing-asp-net-core-in-net-6/> the correct name is **ASP.NET Core in .NET 6**. For brevity, in this book, I will use the term **ASP.NET 6** to highlight an innovation of ASP.NET in .NET 6.

We can create an ASP.NET controller to support the Dapr application's method call, as you can see from the `OrderController.cs` content:

```
namespace sample.microservice.order.Controllers;
[ApiController]
public class OrderController : ControllerBase
{
    [HttpPost("order")]
    public async Task<ActionResult<Guid>> SubmitOrder(Order
        order, [FromServices] DaprClient daprClient)
    {
        Console.WriteLine("Enter submit order");

        order.Id = Guid.NewGuid();
        foreach (var item in order.Items)
        {
            /* a dynamic type is passed to
             sample.microservice.reservation and not
             the Order in scope of
             sample.microservice.order, you could use DTO
             instead */
```

```
var data = new { sku = item.ProductCode,
               quantity = item.Quantity };
var result = await
    daprClient.InvokeMethodAsync<object,
    dynamic>(HttpMethod.Post,
    "reservation-service", "reserve", data);
Console.WriteLine($"sku:
    {result.GetProperty("sku")} == new quantity:
    {result.GetProperty("quantity")}");
}
Console.WriteLine($"Submitted order {order.Id}");
return order.Id;
}
}
```

In the `SubmitOrder` method signature, you will notice that the parameter with the `DaprClient` type has been injected via ASP.NET; the reason why we had to modify the sequence in `Program.cs` is also to support the dependency injection of the Dapr components.

`daprClient` is used here to invoke another service, but it gives our code access to the rest of the Dapr components, such as publish/subscribe and state stores.

The rest of the code is very basic, with no significant logic and, especially, no state stored anywhere. This part will be developed in *Chapter 5, Introducing State Management*.

Once the `order` method of the Dapr `order-service` application is invoked, it cycles through the received payload to invoke the other application.

In `InvokeMethodAsync<object, dynamic>(HttpMethod.Post, "reservation-service", "reserve", data)`, we can recognize the "reservation-service" application name, the "reserve" method, and the payload. The default `HttpMethod` property value for a request is `HttpMethod.Post`; as we are sending a POST request, we could even avoid configuring code this way. If it were a GET request, this would have been necessary.

Our Dapr **Order** service is complete; we can now create the **Reservation** service.

## Creating a project for the Reservation service

We can now add the project for the **Order** service to a solution and create a new project for the **Reservation** service.

We also add the same package reference to the Dapr SDK as we did in the previous project, and we are ready to configure the debugging configuration.

## Preparing the debugging configuration

I followed the debugging instructions from *Chapter 2, Debugging Dapr Solutions*, to create a compound launch configuration in **Visual Studio Code (VS Code)** for the projects corresponding to the two Dapr applications. In the samples, you will find `launch.json` and `tasks.json` properly configured.

I also included the `tye.yaml` configuration. Please note that in this sample, we will directly launch the applications via the Dapr CLI.

Next, we will develop the second ASP.NET Dapr service.

## Implementing Dapr with an ASP.NET Minimal API

In ASP.NET 6, you can control the routing of requests to controllers via attributes and conventional routing. Alternatively, a new approach to API design can be adopted: **Minimal APIs**.

Minimal API is a new template with a simplified approach to creating APIs in ASP.NET, with minimal files and dependencies.

### Learn More about Minimal APIs

While I am used to relying on controllers, the approach we have used so far in this book's samples, the new .NET 6 feature of minimal APIs is especially suited for microservices, thanks to its simplicity.

To learn more about minimal APIs, see <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis> and this excellent exploration on Scott Hanselman's blog: <https://www.hanselman.com/blog/minimal-apis-at-a-glance-in-net-6>.

With the following command, we can create a new service project:

```
C:\Repos\practical-dapr\chapter04>dotnet new webapi -minimal -o sample.microservice.reservation
```

By following the minimal API approach, this is what our **Reservation** service could look like:

```
var builder = WebApplication.CreateBuilder(args);

var jsonOpt = new JsonSerializerOptions()
{
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
    PropertyNameCaseInsensitive = true,
};
```

```
// Add services to the container.
... omitted ...
builder.Services.AddDaprClient(opt => opt.
    UseJsonSerializationOptions(jsonOpt));

var app = builder.Build();
... omitted ...
//app.UseHttpsRedirection();
app.MapPost("/reserve", ([FromServices] DaprClient client,
    [FromBody] Item item) =>
{
    app.Logger.LogInformation("Enter Reservation");
    ... omitted ...
    app.Logger.LogInformation($"Reservation of {storedItem.SKU}
is now {storedItem.Quantity}");
    return Results.Ok(storedItem);
});
app.Run();
```

Starting from the top of `Program.cs`, the code first deals with the ASP.NET configuration and startup, all in a single file – one of the new simplifications brought by .NET 6. Among other things, please note that `DaprClient` gets added to the `WebApplicationBuilder` services.

With the `WebApplication` instance `app` built, we can directly define the method of our Dapr service – in `app.MapPost("/reserve", ([FromServices] DaprClient client, [FromBody] Item item))`, we can appreciate the simplicity by which the method is configured to obtain access to `DaprClient` and deserialize the `Item` type from the request body.

We are now ready to start the testing and debugging of our newly created project – will it work?

Instead of launching the two applications via VS Code, this time we will start the applications with the Dapr CLI from separate terminal windows with these commands, each to be executed in their respective project folders. We first run the “order-service” application:

```
dapr run --app-id "order-service" --app-port "5001" --dapr-
grpc-port "50010" --dapr-http-port "5010" -- dotnet run
--urls="http://+:5001"
```

We then run the “reservation-service” application:

```
dapr run --app-id "reservation-service" --app-port "5002"
--dapr-grpc-port "50020" --dapr-http-port "5020" -- dotnet run
--urls="http://+:5002"
```

While working in self-hosted mode, as in our development environment, we have to carefully avoid port conflicts.

The output of the Dapr CLI can be verbose, though interesting, so for the sake of brevity, these are the lines that tell us that the Order service launch has been successful and is ready to be invoked:

```
PS C:\Repos\practical-dapr\chapter04\sample.microservice.
order> dapr run --app-id "order-service" --app-port "5001"
--dapr-grpc-port "50010" --dapr-http-port "5010" -- dotnet run
--urls="http://+:5001"
Starting Dapr with id order-service. HTTP Port: 5010. gRPC
Port: 50010
... omitted ...
Updating metadata for app command: dotnet run
--urls=http://+:5001
You're up and running! Both Dapr and your app logs will appear
here.
```

Another confirmation that both applications are up and running comes from the `dapr list` output:

```
PS C:\Windows\System32> dapr list
```

APP ID	HTTP PORT	GRPC PORT	APP
order-service	5010	50010	5001
reservation-service	5020	50020	5002

```
run --urls... 4m 2020-09-12 15:31.56 14280
run --urls... 4m 2020-09-12 15:32.09 25552
```

I can now post a JSON payload to the `http://localhost:5010/v1.0/invoke/order-service/method/order` Dapr URL and see the result, and then verify the traces in each of our application’s terminal windows.

When I'm dealing with a JSON payload, I prefer to use Postman or the VS Code REST Client extension (<https://marketplace.visualstudio.com/items?itemName=humao.rest-client>), which offers a better experience, even if we are not using VS Code for debugging:

```
POST http://localhost:5010/v1.0/invoke/order-service/method/
  order HTTP/1.1
content-type: application/json
{
  "Items": [
    {
      "ProductCode": "cookie1",
      "Quantity": 3
    },
    {
      "ProductCode": "ultimate-cookie5",
      "Quantity": 2
    }
  ]
}
```

We receive back a new order ID from the order-service Dapr application:

```
HTTP/1.1
200 OK
Connection: close
Date: Tue, 13 Oct 2020 17:54:20 GMT
Content-Type: application/json;
charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked
"15586359-7413-4838-afde-9adb57ce2d4b"
```

The following output is from the order-service application:

```
== APP == Enter submit order
== APP == Submitted order 15586359-7413-4838-afde-9adb57ce2d4b
```

Yes, that is our newly submitted order ID!

```
This is the reservation-service output:
== APP == Enter Reservation
== APP == Reservation of cookie1 is now -3
== APP == Enter Reservation
== APP == Reservation of ultimate-cookie5 is now -2
```

For each of the items in the submitted order, order-service invokes a reservation-service method.

## Recap

At this stage, any change is ephemeral as no state is preserved. Nevertheless, we successfully built two Dapr services with ASP.NET, using the controller and basic routing approaches, and using `DaprClient`, made available via dependency injection, we established a communication channel via Dapr between the two microservices.

The next section is dedicated to HTTP and gRPC service invocation, which will make things considerably more interesting and complex.

## Comparing HTTP and gRPC for Dapr

gRPC is a *high-performance, open source, universal RPC framework* that became extremely popular in the inter-microservices communication space because of its efficiency. gRPC leverages HTTP/2 for transport and adopts the binary serialization format Protobuf.

### gRPC in .NET

There are several documents and blog posts on how to implement gRPC servers in Dapr, including articles from the .NET documentation on gRPC, <https://docs.microsoft.com/en-us/aspnet/core/tutorials/grpc/grpc-start?tabs=visual-studio-code>, and the (not specific to .NET) Dapr documentation on the gRPC service, <https://docs.dapr.io/operations/configuration/grpc/>.

gRPC is a **Cloud Native Computing Foundation (CNCF)** incubating project.

Let's explore how we can leverage gRPC in ASP.NET and then how to apply it to Dapr.

## gRPC in ASP.NET

In ASP.NET, we are so used to the combination of HTTP and JSON that we tend to think of these two as the only transport and format choices for a web API.

With the growth in popularity of gRPC and the ability to use it from ASP.NET too, we can see it as an alternative for a web API; gRPC requires HTTP/2 with the payload serialized in Protobuf format. For more information, I suggest you read this introduction: <https://grpc.io/docs/what-is-grpc/introduction/>.

In the context of .NET, this document, <https://docs.microsoft.com/en-us/aspnet/core/grpc/comparison>, explains the different perspectives.

## Creating a gRPC microservice

Considering the sample we have built so far in this chapter, we will try to expose the **Reservation** service with gRPC and also instruct the Dapr sidecar of the `reservation-service` application to communicate over gRPC on the last leg of the communication. In addition, we will create a modified copy of the `order-service` application to compose a request with the Protobuf payload. These are the steps for it:

1. Create a new **Reservation** service to be exposed via gRPC.
2. Create the `.proto` file.
3. Configure the project.
4. Implement the necessary service scope.
5. Create a modified **Order** service to use gRPC and Protobuf.
6. Register `reservation-service` as a Dapr application via gRPC.
7. Test the **Order** service with the evolved **Reservation** service.

If we succeed in changing the implementation of a service and the way Dapr interacts with it, we will also demonstrate how Dapr makes it easy to build microservices that are autonomous, as they can have an independent evolution without impacting any other microservice. So, let's understand each of these in the next sections.

### *Creating reservation-service*

As a starting point, create a new .NET project with a gRPC template with the following command-line command:

```
dotnet new grpc -o sample.microservice.reservation-grpc
```

The project is now ready in the `<repository path>\chapter04\ sample.microservice.reservation-grpc\` working area.

### ***Creating the proto file***

We need to describe how messages will be formatted in Protobuf and exchanged via gRPC, as prepared in the `\Protos\data.proto` file:

```
syntax = "proto3";
option csharp_namespace = "sample.microservice.reservation_
    grpc.Generated";
message Item {
    string SKU = 1;
    int32 Quantity = 2;
}
```

The .NET SDK for Dapr already includes the autogenerated proto client.

### ***Configuring the project***

To support gRPC, we should reference several specific packages with the following commands:

```
dotnet add package Grpc.Net.Client
dotnet add package Google.Protobuf
dotnet add package Grpc.Tools
dotnet add package Google.Api.CommonProtos
```

As with any other Dapr application, we also need to reference the Dapr package for ASP.NET with the following command:

```
dotnet add package Dapr.AspNetCore
```

Finally, we reference the proto files we previously created. The following is the configuration snippet to add to the `sample.microservice.reservation-grpc.csproj` project file:

```
<ItemGroup>
    <Protobuf Include="Protos\*.proto" ProtoRoot="Protos"
        GrpcServices="None" />
</ItemGroup>
```

Our project is now properly configured with packages and proto files.

## Service implementation

A gRPC service has some differences from the `Program.cs` file:

```
var builder = WebApplication.CreateBuilder(args);
var jsonOpt = new JsonSerializerOptions()
{
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
    PropertyNameCaseInsensitive = true,
};
// Add services to the container.
builder.Services.AddGrpc();
builder.Services.AddDaprClient(opt => opt.
UseJsonSerializationOptions(jsonOpt));
var app = builder.Build();
//app.UseHttpsRedirection();
app.MapGrpcService<ReservationService>();
app.MapGet("/", () => "Communication with gRPC endpoints must
    be made through a gRPC client...");
app.Run();
```

In the preceding snippet, note that with `builder.Services.AddGrpc()`, the gRPC service has been added to `WebApplicationBuilder`.

The following is the gRPC implementation specific to a Dapr service in `\Services\ReservationService.cs`:

```
namespace sample.microservice.reservation_grpc;

public class ReservationService : AppCallback.AppCallbackBase
{
    private readonly ILogger<ReservationService> _logger;
    private readonly DaprClient _daprClient;
    public ReservationService(DaprClient daprClient,
        ILogger<ReservationService> logger)
    {
        _daprClient = daprClient;
        _logger = logger;
    }
    public override async Task<InvokeResponse>
```

```

OnInvoke(InvokeRequest request, ServerCallContext context)
{
    Console.WriteLine($"Method {request.Method}");
    var response = new InvokeResponse();
    switch (request.Method)
    {
        case "reserve":
            var input = request.Data.Unpack<Generated.
                Item>();
            var output = await Task.FromResult<Generated.
                Item>(new Generated.Item() {SKU=input.SKU,
                Quantity = - input.Quantity});

            response.Data = Any.Pack(output);
            break;
        default:
            Console.WriteLine("Method not supported");
            break;
    }
    return response;
}

public override Task<ListInputBindingsResponse>
    ListInputBindings(Empty request, ServerCallContext
context)
{
    return Task.FromResult(new
        ListInputBindingsResponse());
}

public override Task<ListTopicSubscriptionsResponse>
    ListTopicSubscriptions(Empty request,
        ServerCallContext context)
{
    return Task.FromResult(new
        ListTopicSubscriptionsResponse());
}
}

```

The `ReservationService` class inherits the server-side `Dapr.AppCallback.AutoGen.Grpc.v1.AppCallback.AppCallbackBase` base class and implements the necessary methods.

Our embarrassingly simple code (for which you should leverage a mediator) implements the `OnInvoke(InvokeRequest request, ServerCallContext context)` method and returns an `InvokeResponse` type.

### ***Creating a copy of order-service***

We want to use the gRPC methods in the Dapr client and pass a Protobuf payload to our new `reservation-service` application instead of JSON.

One option is to modify the existing implementation of the `order-service` application; I preferred to create a copy of the `sample.microservice.order` project in the work area, renamed `sample.microservice.order-grpc-client`.

As this project leverages many of the gRPC packages, we should apply the same configuration used in `sample.microservice.order`. The following is a snippet of the `sample.microservice.order-grpc-client.csproj` project file:

```
<ItemGroup>
  <Protobuf Include="..\sample.microservice.reservation-
    grpc\Protos\*.proto" ProtoRoot="..\sample.microservice.
    reservation-grpc\Protos\" GrpcServices="None" />
</ItemGroup>
... omitted ...
<ItemGroup>
  <PackageReference Include="Google.Protobuf"
    Version="3.19.4" />
  <PackageReference Include="Google.Api.CommonProtos"
    Version="2.5.0" />
  <PackageReference Include="Grpc.Tools"
    Version="2.43.0" PrivateAssets="All" />
</ItemGroup>
```

`OrderController.cs` requires a change in code to leverage the gRPC methods provided by `DaprClient`:

```
[HttpPost("order")]
public async Task<ActionResult<Guid>> SubmitOrder(Order
order, [FromServices] DaprClient daprClient)
{
    Console.WriteLine("Enter submit order");
    order.Id = Guid.NewGuid();
```

```
foreach (var item in order.Items)
{
    var data = new sample.microservice.
        reservation_grpc.Generated.Item()
    { SKU = item.ProductCode,
      Quantity = item.Quantity };
    var result = await daprClient.
        InvokeMethodGrpcAsync<sample.microservice.
            reservation_grpc.Generated.Item,
            sample.microservice.reservation_grpc.
                Generated.Item>("reservation-service",
                "reserve", data);
    Console.WriteLine($"sku: {result.SKU} ==
        new quantity: {result.Quantity}");
}
Console.WriteLine($"Submitted order {order.Id}");
return order.Id;
}
}
```

The client code is ready to invoke the `daprClient.InvokeMethodGrpcAsync` method to exchange the Protobuf payload via gRPC with the service, via the Dapr infrastructure.

### ***Test integration***

In order to launch our service as a Dapr application, we have to change the way we use the Dapr CLI:

```
PS C:\Repos\practical-dapr\chapter04\sample.microservice.
reservation-grpc> dapr run --app-id "reservation-service"
--app-port "3000" --app-protocol grpc --dapr-grpc-port "50020"
-- dotnet run --urls="http://+:3000"
Starting Dapr with id reservation-service. HTTP Port: 55614.
gRPC Port: 50020
```

```

... omitted ...
== APP ==          Request starting HTTP/2 POST
http://127.0.0.1:3000/dapr.proto.runtime.v1.AppCallback/
ListInputBindings application/grpc
== APP ==          Executing endpoint 'gRPC - /dapr.proto.runtime.
v1.AppCallback/ListInputBindings'
== APP ==          Executed endpoint 'gRPC - /dapr.proto.runtime.
v1.AppCallback/ListInputBindings'
== APP ==          Request starting HTTP/2 POST
http://127.0.0.1:3000/dapr.proto.runtime.v1.AppCallback/
ListTopicSubscriptions application/grpc
... omitted ...
== APP ==          Executing endpoint 'gRPC - /dapr.proto.runtime.
v1.AppCallback/ListTopicSubscriptions'
== APP ==          Executed endpoint 'gRPC - /dapr.proto.runtime.
v1.AppCallback/ListTopicSubscriptions'
You're up and running! Both Dapr and your app logs will appear
here.

```

In the `dapr run --app-id "reservation-service" --app-port "3000" --app-protocol grpc --dapr-grpc-port "50020" -- dotnet run --urls="http://+:3000"` command, we specified that Dapr will interact with our service via gRPC with the `--app-protocol` parameter.

We can launch the modified **Order** service:

```

PS C:\Repos\practical-dapr\chapter04\sample.microservice.
order-grpc-client> dapr run --app-id "order-service" --app-port
"5001" --dapr-grpc-port "50010" --dapr-http-port "5010" --
dotnet run --urls="http://+:5001"

```

As `order-service` leverages `DaprClient` from the Dapr .NET SDK, it will reach the evolved `reservation-service` instance using gRPC at each step.

## Winning latency with gRPC

In this scenario, we have to consider the potential impact of a minuscule latency gain, obtained with a service-to-service communication fully made with gRPC:

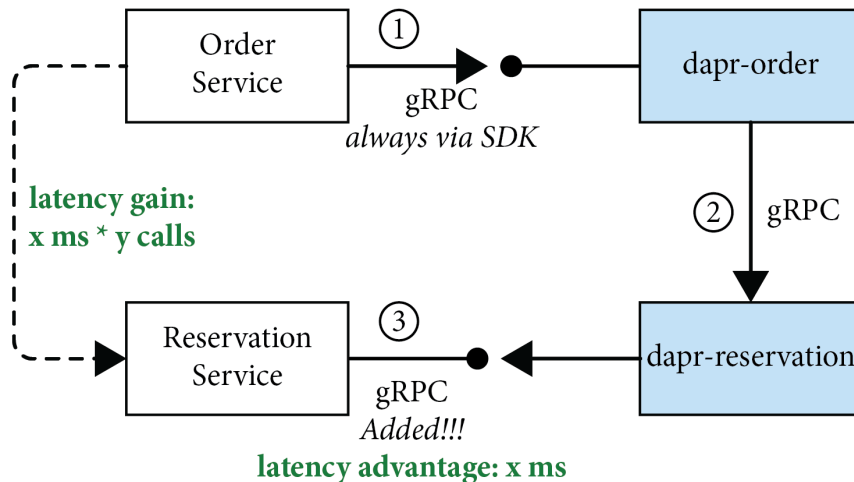


Figure 4.4 – Service-to-service invocation via gRPC

Figure 4.4 depicts how each small advantage can accrue significant latency reductions with large numbers of requests.

It's not this chapter's objective to discuss the pros and cons of gRPC, nor to convince you to favor exposing microservices to Dapr as gRPC services. My perspective is that in most cases, the advantages in productivity that come with the HTTP and JSON approach in the ASP.NET web API overcome the marginal latency gains with gRPC in the last leg, considering that Dapr already grants the benefit of gRPC inter-node latency.

We will dig deeper into the microservices paradigm by addressing the elephant in the room – that is, managing state within Dapr – in the next chapter.

## Summary

This intense chapter introduced you to the most relevant building block of Dapr – service-to-service invocation.

We learned how to configure an ASP.NET project, how to decorate methods in a controller with Dapr's specific parameters, and how to interact from one Dapr service to another.

Finally, we discussed the more complex mechanics of a gRPC service in the context of Dapr, which could be helpful in some scenarios, at the cost of increased complexity by using Dapr.

You may have noticed that our sample microservices are not persisting in their state yet. In the next chapter, we will explore how to use the Dapr system's state management building block.

## Questions

1. What is new in .NET 6 for ASP.NET?
2. Do you need to configure Dapr to use gRPC?
3. Do you need gRPC for your microservices?
4. Which is better: HTTP or gRPC?

## Further reading

- *How-To: Invoke services using gRPC*: <https://docs.dapr.io/developing-applications/building-blocks/service-invocation/howto-invoke-services-grpc>
- *Using Dapr with gRPC* by Donovan Brown: <https://www.youtube.com/watch?v=Q2xS153IofM>
- *Minimal APIs overview*: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis?view=aspnetcore-6>



# Introducing State Management

State management for services and actors is a centerpiece of Dapr. This chapter will illustrate how an application can transfer the responsibility of managing state to Dapr, with the ability to switch between different state store types.

These are the main topics that we will explore:

- Managing state in Dapr
- Stateful services in an e-commerce ordering system
- Using **Azure Cosmos DB** as a state store

Most, if not all, of our services and actors in the Dapr applications we are building have data persisted as a state.

The state could be the status of a request, kept aside to be able to return additional information of a complex interaction at a later stage, or it could be the central information managed by the service, such as the quantity of the available product.

State management is equally important for a new, cloud-native solution built with Dapr and for an existing solution to which we are adding Dapr services.

An overview of state management concepts is our starting point.

## Technical requirements

The code for this chapter can be found on GitHub at <https://github.com/PacktPublishing/Practical-Microservices-with-Dapr-and-.NET-Second-Edition/tree/main/chapter05>.

In this chapter, the working area for scripts and code is expected to be `<repository path>\chapter05\`. In my local environment, it is `C:\Repos\practical-dapr\chapter05`.

Please refer to the section entitled *Setting up Dapr* in *Chapter 1, Introducing Dapr*, for a complete guide on the tools needed to develop with Dapr and work with the samples.

## Managing state in Dapr

In a microservice architecture, **state** refers to the collection of information that defines the context in which the microservice operates. In this section, we will learn how state can be managed and how Dapr does it.

### State, stateless, and stateful

The way that state is managed defines whether a microservice is **stateful** (when it takes the responsibility of persisting the state upon itself) or **stateless** (when the state is not in its scope of responsibility).

An example of a stateful microservice would be a shopping cart microservice that keeps the list of items in a central location (such as a database) so the customer can transparently migrate between different devices to continue their shopping experience. The shopping cart microservice could be designed by keeping the state in the host/node memory and enforcing a policy at the load balancer level to route all further interactions from the client to the original node.

Would it be a good idea, in the age of the cloud, to adopt this approach? Simply not. To gain a higher level of resiliency in modern architecture, the nodes should be considered expendable resources that can fail without impacting the solution.

A good example of a stateless microservice is a machine learning model exposed via an API, which, given a set of input parameters, would return a prediction. A microservice supporting a business activity is likely to count on a state. As the microservice should be able to scale to additional instances or restore from process or host failures, it becomes increasingly relevant to keep the state externally, for example, on a **Database as a Service (DBaaS)** offering, relieving our architecture of these hard-to-solve problems. The shopping cart microservice could also adopt a stateless approach by persisting the state on an external store.

With the reliable state API provided by Dapr, every microservice built with the Dapr runtime can be considered a stateless service, as it keeps its state in an external, configurable store.

Let's explore some of the state store options Dapr offers to developers.

### State stores in Dapr

A **state store** is a Dapr component that implements reliable store interfaces.

There are plenty of state stores in Dapr; several provide support for the major cloud platforms, such as Microsoft Azure, and database services, while many others support generic databases you could run in the cloud or on the edge.

### Certification life cycle

Each component in Dapr goes through a certification life cycle, thoroughly described at <https://docs.dapr.io/operations/components/certification-lifecycle/>. The **Alpha**, **Beta**, and **Stable** certification levels show the maturity of a component, giving you the ability to make an informed decision on which component to choose for your use case.

The following are only a few of the state stores available in Dapr:

- Azure Cosmos DB
- Azure Blob storage
- Azure SQL Server
- Apache Cassandra
- MongoDB
- MySQL
- PostgreSQL
- Redis

The open and extensible capabilities of Dapr allow any party to implement additional state stores. You can find the most up-to-date list of state components in the Dapr documentation at <https://docs.dapr.io/reference/components-reference/supported-state-stores/>.

Multiple stores can be configured in the same hosting environment, enabling your application to retrieve, update, or delete the state as a key/value (JSON-serializable) pair.

In your local development environment, the Redis state store is configured by default, pointing to a Redis container deployed during the `dapr init` setup.

Each state store, depending on the database or service capabilities, may provide additional features. Let's explore transaction support first, then concurrency, and finally consistency.

## Transactions

A Dapr state store can coordinate the queries to the database, resulting from the application's interaction with the Dapr state management building block, under a transactional context by implementing the `TransactionalStore` interface.

Only a subset of state stores in Dapr support transactions, such as the following ones:

- Azure Cosmos DB
- Azure SQL Server

- MongoDB
- Redis
- PostgreSQL

A specific scenario requiring transaction support involves the **actor model** in Dapr. We will explore this in more depth in *Chapter 8, Using Actors*.

## Concurrency

Dapr gives developers control over concurrency in state manipulation by returning and accepting an ETag. An **ETag** is metadata used to identify the version of a resource, in this case, a key/value pair in Dapr state management.

An application retrieving the state can retain the attached ETag and later re-attach it to an update request, in order to prevent overwriting a newer state that might have changed in the interim. If the ETag sent back to Dapr does not match the original one, the operation is rejected as the first write wins.

By using the Dapr **C# SDK**, the ETag can be managed transparently for you while dealing with state management. If no ETag is provided in the state change request, a last-write-wins approach is applied. To learn more about the Dapr approach to concurrency, please see the documentation at <https://docs.dapr.io/developing-applications/building-blocks/state-management/state-management-overview/#concurrency>.

If you foresee a scenario in which your service will experience concurrent service requests, with an ETag, your application can make sure unintended state overwrites are avoided.

## Consistency

Consistency in state change requests can also be controlled by the application. If eventual consistency is preferred (this is the default), the state change is considered successful by Dapr as soon as the underlying state store acknowledges the write operation. If strong consistency is required instead by the application, Dapr waits for the state store to complete the write operation on all of its replicas.

Not all state stores support both eventual and strong consistency modes; Azure Cosmos DB is one of those that does. You can learn more about the performance impact of consistency in the documentation available at <https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels>.

By controlling the consistency, your application code can specify, for each operation, the risks versus benefits of retrieving and persisting a state.

As we explored state store capabilities, let's see how applications can leverage them.

## Interaction with state stores

An application can directly invoke the Dapr sidecar via the `http://localhost:<daprPort>/v1.0/state/<storename>/<key>` HTTP call or you can leverage the abstraction offered by the SDK.

The following diagram depicts the interaction between your application and the Dapr sidecar, which, influenced by the component's configuration, leverages a state store:

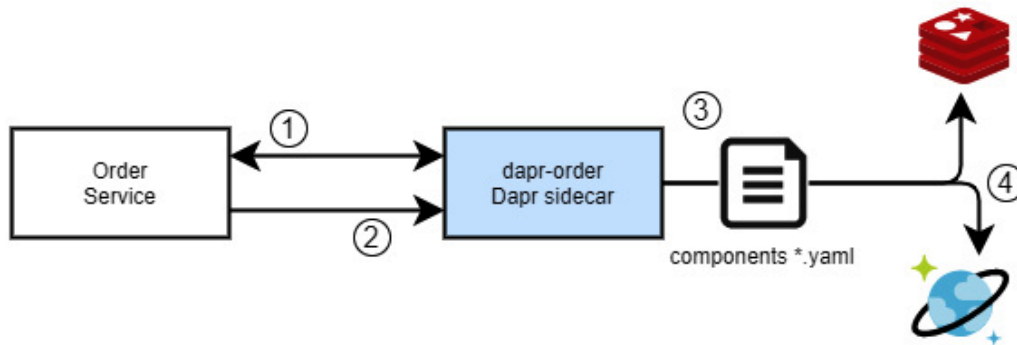


Figure 5.1 – State stores in Dapr

The diagram in *Figure 5.1* describes the steps of a state operation in Dapr. Let's explore them in detail:

1. The application invokes the local URL provided by the Dapr sidecar, for example, GET `http://localhost:<daprPort>/v1.0/state/shoppingcart/43`, to retrieve the state.
2. The Dapr state can be updated by an application with an HTTP POST request to the Dapr state API, `http://localhost:<daprPort>/v1.0/state/shoppingcart/43`. The state can also be deleted with a DELETE request to the same Dapr endpoint.
3. The state stores available to the Dapr application are defined via `.yaml` files, present in a `components` folder, or applied to the Kubernetes configuration.

To accommodate the request, a component named `shoppingcart` is expected to be configured.

4. While locally, Dapr could be using the Redis local container provided by Dapr, on Kubernetes, it could be relying on an external state such as Azure Cosmos DB. All we need is a change to the component's `.yaml` files.

The key of a Dapr state uses the same format on all state stores, by combining the application ID and the state key used in the application with the `<App ID> || <state key>` format. Following the previous example, the key for the persisted record of the shopping cart would be `shoppingcart || 43`.

Microservice architecture suggests keeping state, and data in general, separate. Nevertheless, with this approach, Dapr lets us use the same configured state store with different Dapr applications without the risk of key collisions. 43 in the context of `shoppingcart` will have a different composed key than the state record of order number 43.

The following is a YAML description of a component that uses the local Redis:

```
apiVersion: daprio/v1alpha1
kind: Component
metadata:
  name: statestore
spec:
  type: state.redis
  version: v1
  metadata:
    - name: redisHost
      value: localhost:6379
    - name: redisPassword
      value: ""
```

In the previous snippet, a component of the `state.redis` type has been configured to connect to the Redis local instance.

## Resiliency

As we learned the basics of how an application interacts with the state store via the Dapr sidecar, it is worth briefly exploring the impact of the resiliency feature on the state management building block.

The resiliency feature allows developers and operators to configure the way Dapr applications handle transient errors. As we learned in the *Resiliency* section in *Chapter 4, Service-to-Service Invocation*, there are a few types of policies that can be applied to targets such as a state component.

Why would we want to apply a resiliency policy to a state store? An example could be to gracefully handle a transient connection error to the database, which is well described in this guide on cloud development best practices: <https://docs.microsoft.com/en-us/azure/architecture/best-practices/transient-faults>.

Resiliency is also important to manage the stress our Dapr application puts onto the state store.

If the database we selected for the state store of our Dapr application starts to experience errors, maybe because of excessive requests to load, how do we want to handle it? Would a constant retry policy help? It depends on many factors, but it might risk aggravating the situation by flooding the state store with repeated requests.

By using a circuit breaker policy instead, as soon as errors exceed a configured threshold, interactions are suspended for a timeout, only to fully resume after a limited number of operations have been successful.

The next section is dedicated to how a stateful service for a common e-commerce platform can be organized with Dapr.

## Stateful services in an e-commerce ordering system

Moving forward with the *Order-Reservation* scenario introduced in the previous chapters, at this stage, we will focus on persisting the state in each Dapr application.

The following diagram anticipates the change in state management that we are going to apply to our microservices:

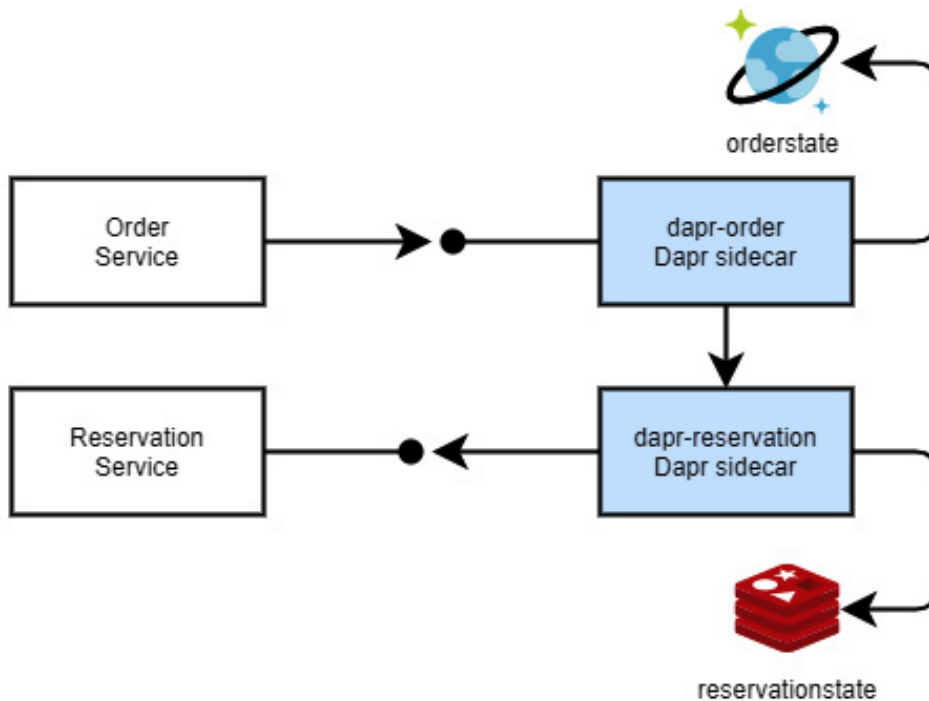


Figure 5.2 – Multiple state stores in Dapr

As you can see in *Figure 5.2*, the Dapr reservation-service service is going to use Redis as the state store, while order-service is going to leverage the Cosmos DB state store.

The following are the project structures used to support the `order-service` and `reservation-service` Dapr applications:

- `sample.microservice.dto.order`
- `sample.microservice.order`
- `sample.microservice.dto.reservation`
- `sample.microservice.reservation`

I decided to have **Data Transfer Object (DTO)** libraries that a service client can use to interact with the service itself, separate from the objects used to persist the state. This is just one of many possible approaches.

We'll start by implementing state management in `reservation-service`.

## Stateful reservation-service

What makes the Dapr application's `reservation-service` a stateful service? The short answer is that the application keeps track of the balance quantity for items, evaluating the reservation request originating from the received orders.

`order-service` does not keep track of the item balance between one reservation and the next. Instead, it relies on `reservation-service` to manage this information and preserve it safely over time; this is what is generally expected from a stateful service.

A client interacting with the **ASP.NET** service endpoint would use `POST http://localhost:5002/reserve` to reserve the quantity of a product or `GET http://localhost:5002/balance/cookie2` to retrieve the current balance of a specific product (with 5002 being the port used by ASP.NET), while a different Dapr application would invoke the local sidecar (with 5020 being the port used to reach Dapr) at `POST http://localhost:5020/v1.0/invoke/reservation-service/method/reserve` to ultimately reach the `reservation-service` Dapr application, as we learned in *Chapter 4, Service-to-Service Invocation*.

Regardless of the route and approach used, the important point here is that Dapr enable us to code `reservation-service` like a stateless service as all the complexities of managing the state are transferred to the Dapr sidecar and the configured state store.

## Handling the Dapr state in ASP.NET

The `DaprClient` class gives our code access to the Dapr runtime by abstracting the interaction with the API endpoint exposed by the sidecar.

An instance is made available to the `Controller` method via dependency injection. To leverage minimal hosting in **.NET 6**, we need to add `.AddDapr` to `Program.cs`:

```

... omitted ...
var jsonOpt = new JsonSerializerOptions()
{
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
    PropertyNameCaseInsensitive = true,
};
// Add services to the container.
builder.Services.AddControllers().AddDapr(opt => opt.
UseJsonSerializationOptions(jsonOpt));
... omitted ...

```

In the following code snippets, you will see how `daprClient` gives us easy access to the overall Dapr runtime including states, services, and messages.

We can examine the code in `ReservationController.cs`:

```

namespace sample.microservice.reservation.Controllers;
[ApiController]
public class ReservationController : ControllerBase
{
    public const string StoreName = "reservationstore";
    [HttpPost("reserve")]
    public async Task<ActionResult<Item>> Reserve(Item
        reservation, [FromServices] DaprClient daprClient)
    {
        var state = await daprClient.GetStateEntryAsync
            <ItemState>(StoreName, reservation.SKU);
        ... changes to state ...
        Console.WriteLine($"ETag {state.ETag}");
        var saved = await state.TrySaveAsync();
        if (saved != true)
        {
            Console.WriteLine("Failed to save state");
            return this.StatusCode(500);
        }
        var result = new Item() {SKU = state.Value.SKU,
            Quantity= state.Value.BalanceQuantity};
        Console.WriteLine($"Reservation of {result.SKU} is now

```

```
        {result.Quantity}");  
        return result;  
    }  
    ... omitted ...
```

In the previous code snippet, we have the `[HttpPost("reserve")]` attribute establishing a route that the Dapr runtime will leverage to invoke the ASP.NET controller.

The signature of the async `Reserve` method returns a result of the `Task<ActionResult<Item>>` type, with `Item` being a DTO type. The same type is accepted from clients as it is used for the input parameter.

The state is requested, explicitly in the method code, asynchronously to the Dapr sidecar with `await daprClient.GetStateEntryAsync<ItemState>(StoreName, reservation.SKU)`, in which the two parameters are `storeName`, which is the name of the configured state store, and the key to look for in the state store.

The method to save the state back to the Dapr store used here is `state.TrySaveAsync()`, which implicitly leverages the ETag. As previously described, by passing an ETag, we take a first-write-wins approach; we can be sure that no other changes occurred to this specific state entry while processing the request.

Slightly different is the approach for the `GET` method implemented in `ReservationController.cs`:

```
... omitted ...  
[HttpGet("balance/{state}")]  
public ActionResult<Item> Get([FromState(StoreName)]  
    StateEntry<ItemState> state)  
{  
    Console.WriteLine("Enter item retrieval");  
    if (state.Value == null)  
    {  
        return this.NotFound();  
    }  
    var result = new Item() {SKU = state.Value.SKU,  
        Quantity= state.Value.BalanceQuantity};  
    Console.WriteLine($"Retrieved {result.SKU} is {result.  
        Quantity}");  
    return result;  
}  
}
```

The `[HttpGet("balance/{state}")]` attribute influences the routing of requests to this method, and the state gets implicitly requested to Dapr via the `[FromState(StoreName)]` attribute of `StateEntry<ItemState> state`; an instance of the `StateEntry` type, with a value of the `ItemState` type, is retrieved from the state store with the key passed in with `balance/{state}`.

It could be the case that there is no state registered for the submitted key, which can be evaluated by checking the `state.Value` property for `null` and, eventually, returning a `NotFound` result back to the caller.

For the time being, the `reservationstore` used by `reservation-service` is a state store component pointing to the local Redis; just by changing the component definition, it could switch to another, completely different store option.

So far, we have learned how to use `DaprClient` to manage state in our Dapr applications and how to configure a state store component.

In the next section, we will use Cosmos DB as a state store option and verify the implementation from Dapr's perspective.

## Using Azure Cosmos DB as a state store

Instead of using the local Redis storage, we are going to leverage another Dapr state store option, **Azure Cosmos DB**, a globally distributed, multi-model database service.

The steps needed to configure the new state store are as follows:

1. Setting up Azure Cosmos DB
2. Configuring the state store
3. Testing the state store
4. Partitioning with Cosmos DB
5. Wrapping up

The application code of `reservation-store` will not be changed; we will only operate at the configuration level of the state store component.

We'll start by setting up the Azure Cosmos DB resource.

### Setting up Azure Cosmos DB

To create a Cosmos DB instance on Azure, please follow this guide in the Azure documentation: <https://docs.microsoft.com/en-us/azure/cosmos-db/how-to-manage-database-account>.

You should also take a look at the Dapr documentation for the same purpose: <https://docs.dapr.io/developing-applications/building-blocks/state-management/query-state-store/query-cosmosdb-store/>.

I created an Azure Cosmos DB account and a database named `state`, and then provisioned two containers, one for the `reservation-service` state store and a second one for the `order-service` state store:

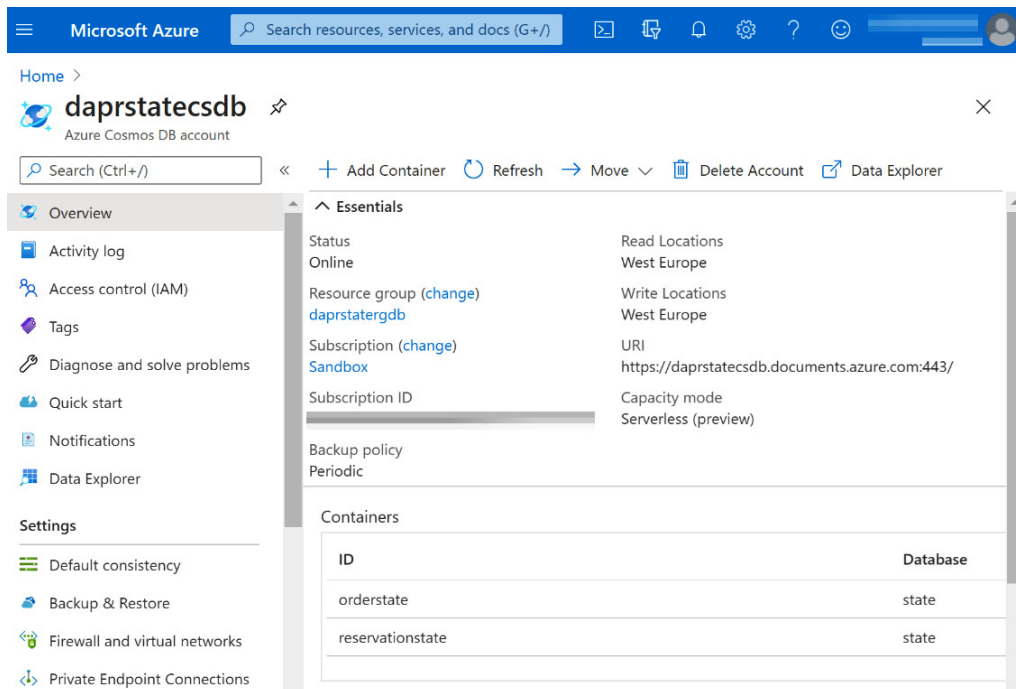


Figure 5.3 – Azure Cosmos DB configured with containers

I am using the serverless service tier of Cosmos DB, which bills only based on usage (it is the best option for a small application that doesn't have sustained traffic).

We are now ready to configure the state store component.

## Configuring the state store

The Dapr component for Cosmos DB is slightly different than the one for Redis, as you can see in the following code block:

```
apiVersion: dapr.io/v1alpha1
kind: Component
```

```
metadata:
  name: orderstore
spec:
  type: state.azure.cosmosdb
  version: v1
  metadata:
    - name: url
      value: https://daprstatedcsdb.documents.azure.com:443/
    - name: masterKey
      value: <omitted>
    - name: database
      value: state
    - name: collection
      value: orderstate
```

url, database, and collection match the provisioned Cosmos DB resources.

#### Important note

It is highly recommended that you do not keep secrets and credentials in configuration files and that you rely on the Dapr secret management feature instead.

During the `dapr run` launch sequence, we can look for evidence that the configured state stores are evaluated. From the lengthy output, the following is an excerpt:

```
PS C:\Repos\practical-dapr\chapter05> dapr run --app-id
"order-service" --app-port "5001" --dapr-grpc-port "50010"
--dapr-http-port "5010" --components-path "./components"
-- dotnet run --project ./sample.microservice.order/
sample.microservice.order.csproj --urls="http://+:5001"
```

The following is the corresponding output:

```
Starting Dapr with id order-service. HTTP Port: 5010. gRPC
Port: 50010
== DAPR == time="..." level=info msg="starting Dapr Runtime
-- version 1.1.0 -- commit 6032dc2" app_id=order-service
instance=DB-XYZ scope=dapr.runtime type=log ver=1.1.0
== DAPR == time="..." level=info msg="log level set to: info"
```

```

app_id=order-service instance=DB-XYZ scope=dapr.runtime
type=log ver=1.1.0
== DAPR == time="..." level=info msg="found component
reservationstore (state.azure.cosmosdb)" app_id=order-service
instance=DB-XYZ scope=dapr.runtime type=log ver=1.1.0
== DAPR == time="..." level=info msg="found component orderstore
(state.azure.cosmosdb)" app_id=order-service instance=DB-XYZ
scope=dapr.runtime type=log ver=1.1.0

```

I added `--components-path "./components"` to specify the location of the `.yaml` file's components, otherwise, Dapr would leverage the default components with Redis.

The following information messages tell us that `reservationstore (state.azure.cosmosdb)` and `orderstore (state.azure.cosmosdb)` have been correctly applied.

To avoid any confusion, `order-service` is going to use the `orderstore` state store, while `reservation-service` will leverage `reservationstore`. The `.yaml` files are located in the same folder.

The following diagram depicts the configuration change we applied:

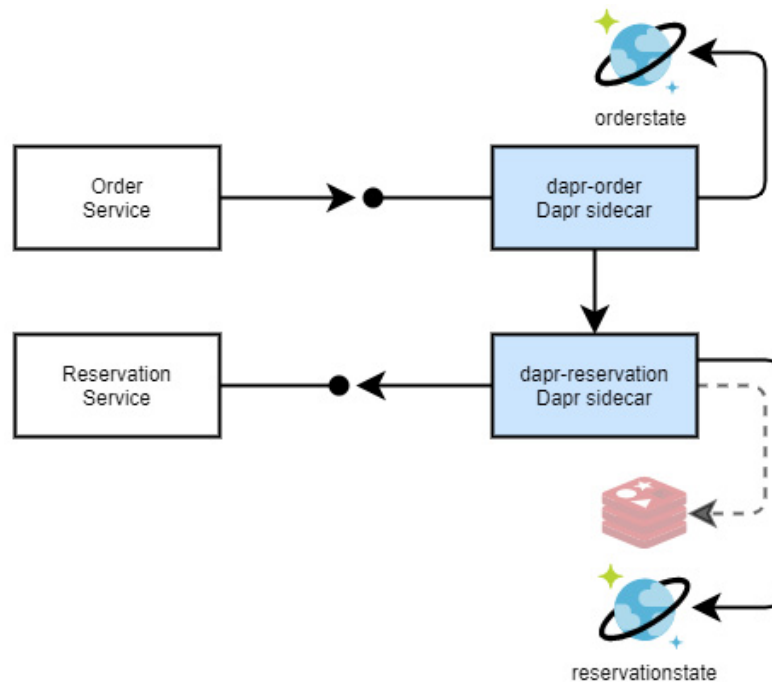


Figure 5.4 – Impact of a Dapr configurable state store

Figure 5.4 depicts the transparent change of the state store used by the `reservation-service` application from Redis to Azure Cosmos DB.

As we configured Cosmos DB as the state and we have confirmation that the component is loaded by Dapr, it is time to test it!

## Testing the state store

With `order-service` and `reservation-service` running and state stores configured, it is time to test the services to appreciate how data is persisted on Azure Cosmos DB.

As provided in the `order.test.http` file, a POST command to the ASP.NET endpoint of the `order-service` endpoint will persist our new item in state, in addition to invoking the `reservation-service` application:

```
POST http://localhost:5001/order
content-type: application/json
{
  "CustomerCode": "Davide",
  "Date": "2022-03-19T08:47:53.1224585Z",
  "Items": [
    {
      "ProductCode": "cookie4",
      "Quantity": 7
    },
    {
      "ProductCode": "bussola1",
      "Quantity": 6
    }
  ]
}
```

The following GET command invokes the `order-service` ASP.NET endpoint, which in turn invokes the Dapr state API:

```
GET http://localhost:5001/order/08ec11cc-7591-4702-bb4d-7e86787b64fe
###
GET http://localhost:5010/v1.0/state/orderstore/08ec11cc-7591-4702-bb4d-7e86787b64fe
```

While the Dapr application relies on the .NET SDK to get and save the state, it is often useful to know how to check the persisted state directly by interacting with the Dapr API.

We can inspect how data is persisted as an item in Azure Cosmos DB, as shown in the following screenshot:

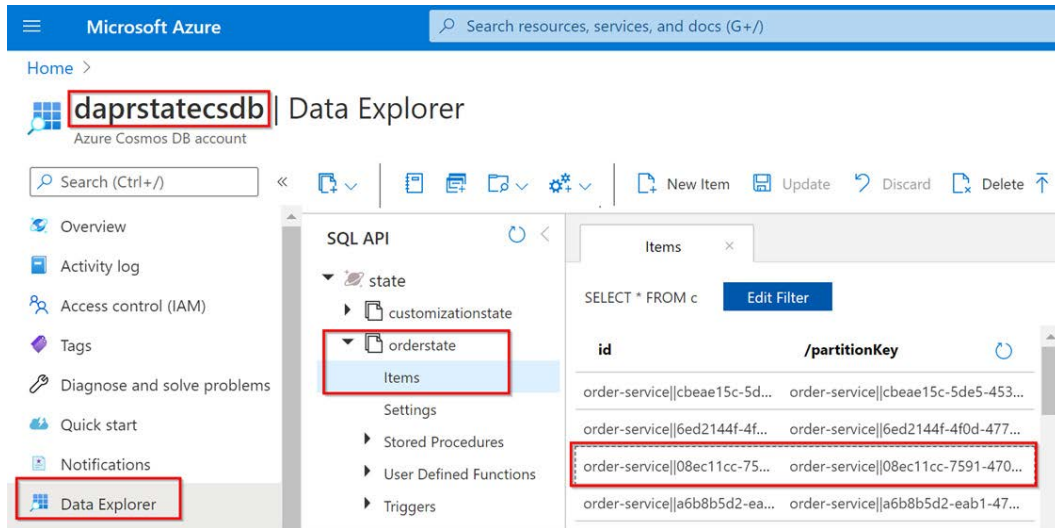


Figure 5.5 – Data Explorer in Azure Cosmos DB

As shown in *Figure 5.5*, we can reach the data persisted by the Dapr state with the following steps:

1. Open **Data Explorer** from the navigation pane
2. Select the database and the relevant container
3. Browse the items
4. Examine the record content containing the Dapr state:

```

1  {
2    "id": "order-service|08ec11cc-7591-4702-bb4d-7e86787b64fe",
3    "value": {
4      "CreatedOn": "2020-09-19T12:28:19.2450723Z",
5      "UpdatedOn": "2020-09-19T12:28:19.2451555Z",
6      "Order": {
7        "Date": "2020-09-19T08:47:53.1224585Z",
8        "Id": "08ec11cc-7591-4702-bb4d-7e86787b64fe",
9        "CustomerCode": "Davide",
10       "Items": [
11         {
12           "ProductCode": "cookie4",
13           "Quantity": 7
14         },
15         {
16           "ProductCode": "bussola1",
17           "Quantity": 6
18         }
19       ]
20     }
21   },
22   "partitionKey": "order-service|08ec11cc-7591-4702-bb4d-7e86787b64fe",
23   "_rid": "h+ETA0zuFXEMAAAAAAAAA==",
24   "_self": "dbs/h+ETAA==/colls/h+ETA0zuFXE=/docs/h+ETA0zuFXEMAAAAAAAAA==/",
25   "_etag": "\"18006115-0000-0d00-0000-5f65f9630000\"",
26   "_attachments": "attachments/",
27   "_ts": 1600518499
28 }

```

Figure 5.6 – Item in Azure Cosmos DB Data Explorer

As shown in *Figure 5.6*, the state is persisted as JSON, with the main payload in the `value` field and the item ID being the Dapr state key composed with the `<application Id> | <state key>` pattern.

We learned firsthand how Cosmos DB Dapr's state store persists our application state in the database, so let's now investigate how the key and distribution of records are correlated.

## Partitioning with Cosmos DB

**Partitioning** enables Azure Cosmos DB to scale individual containers in a database, as items grouped along the partition key in logical partitions are then positioned in physical partitions; it is one of the most relevant concepts that Cosmo DB adopts in terms of offering a high level of performance.

By default, the Dapr state key is used as the `partitionKey` value for each item in the Cosmos DB container. Considering that the Dapr state API always uses the key to read and write data from Cosmos DB, it should be a good choice in most cases.

If you need to influence the value of `partitionKey`, you can specify it with the `metadata` parameter while interacting with the state object. As an example, let's see the following code:

```
var metadata = new Dictionary<string, string>();
metadata.Add("partitionKey", "something_else");
await state.SaveChangesAsync(metadata: metadata);
```

In cases where we modified our `order-service` Dapr application to adopt a different partitioning scheme, this would be the persisted document in Cosmos DB:

```
{
  "id": "order-service|f5a34876-2737-4e8f-aba5-002a4e1ab0cc",
  "value": {
    "CreatedOn": "2020-09-19T11:15:20.7463172Z",
    "UpdatedOn": "2020-09-19T11:15:20.746392Z",
    "Order": {
      "Date": "2020-09-19T08:47:53.1224585Z",
      "Id": "f5a34876-2737-4e8f-aba5-002a4e1ab0cc",
      "CustomerCode": "Davide",
      "Items": [
        {
          "ProductCode": "cookie4",
          "Quantity": 7
        },
        {
          "ProductCode": "bussola1",
          "Quantity": 6
        }
      ]
    }
  },
  "partitionKey": "something_else",
  "_rid": "h+ETAOzuFXELAAAAAAAAAA==",
  "_self": "dbs/h+ETAA==/colls/h+ETAOzuFXE=/docs/h+ETAOzuFXELAAAAAAAAAA==/",
  "_etag": "\"1800590c-0000-0d00-0000-5f65e8480000\"",
  "_attachments": "attachments/"
}
```

```
"_ts": 1600514120
}
```

In the preceding code snippet, you can see "partitionKey": "something\_else" and how it differs from the "id": "order-service||f5a34876-2737-4e8f-aba5-002a4e1ab0cc" key, being influenced by the metadata parameter in the SaveAsync method.

In most cases, the default approach to partitioning should be fine, but now you know how to control it.

Using Azure Cosmos DB as a state store for Dapr does not prevent us from using it for additional scenarios, as we will learn next.

## Wrapping up

As we were able to prove, Dapr influences the data persisted in the state store with its minimalistic approach. Apart from the key/value, the payload is untouched and consumable, just like any other JSON document.

This opens up an additional scenario: *what if I need to search for orders submitted to order-service that contain a specific item by ProductCode?* This is not information that we can search for using the Dapr runtime, as you can see here:

```
SELECT c.id FROM c WHERE ARRAY_CONTAINS(c["value"]["Order"].
Items, {"ProductCode": "bussola2"}, true)
```

The query shown, executed on the container used as the state store by order-service, will return a list of item IDs, including the following:

```
[
  {
    "id": "order-service||bbc1f16a-c7e3-48c3-91fb-
      b2175acfc299"
  },
  {
    "id": "order-service||91705574-df80-4844-af5f-
      66877c436e9b"
  },
  {
    "id": "order-service||ac1e173e-fe4e-476f-b6f6-
      e9615a49f47b"
  }
]
```

A method in our Dapr application could perform the query directly against the Azure Cosmos DB instance using the native .NET SDK to obtain the state key from the ID. As we know, the Dapr state key is composed as `<App ID> | <state key>`, and we can derive the orders in the scope of our search. Also, `order.id` `bbc1f16a-c7e3-48c3-91fb-b2175acfc299`, `91705574-df80-4844-af5f-66877c436e9b`, and `ac1e173e-fe4e-476f-b6f6-e9615a49f47b` are the ones containing the `ProductCode` value we were looking for.

This is not information that we could obtain with a query via the Dapr runtime to the API.

Any requirement outside the scope of the Dapr state API can be approached natively by interacting with, in this scenario, Cosmos DB.

External manipulation of the state should always be avoided, but there is nothing preventing you from reading it.

We have now completed our exploration of a powerful database, Azure Cosmos DB, used in Dapr as a state store.

## Summary

In this chapter, we introduced the state management API provided by Dapr and learned how an ASP.NET service can leverage it via the .NET SDK.

We also appreciated the flexibility offered by Dapr to developers and operators in terms of configuring and modifying the state stores.

By testing the Dapr state management with local Redis and then with the cloud-based Azure Cosmos DB, we proved how easy it is not only to move the state outside of a stateful microservice but also to shift from one persistence technology to another simply via a configuration change.

In the next chapter, we will discuss a flexible and scalable approach to communicating between Dapr applications.

# 6

## Publish and Subscribe

**Publish/Subscribe (pub/sub)** is a messaging pattern supported by **Distributed Application Runtime (Dapr)** to enable decoupled interactions between microservices.

In the previous chapters, we examined how applications can communicate with one another via direct service-to-service invocation. In this chapter, you will learn about the benefits of the pub/sub messaging-based pattern and how to implement it in your Dapr applications.

We have the following as the main topics of this chapter:

- Using the pub/sub pattern in Dapr
- Using **Azure Service Bus (ASB)** in Dapr
- Implementing the saga pattern

Before we delve into the implementation details of pub/sub in Dapr, an overview of the pattern is necessary.

### Technical requirements

The code for this sample can be found on GitHub at <https://github.com/PacktPublishing/Practical-Microservices-with-Dapr-and-.NET-Second-Edition/tree/main/chapter06>.

In this chapter, the working area for scripts and code is expected to be `<repository path>\chapter06\`. In our local environment, it is `C:\Repos\practical-dapr\chapter06`.

Please refer to the *Setting up Dapr* section in *Chapter 1, Introducing Dapr*, for a complete guide to the tools needed to develop with Dapr and work with the samples.

### Using the pub/sub pattern in Dapr

In microservice architectures, the **pub/sub** pattern is widely adopted to facilitate the creation of a decoupled communication channel between different parts of an application.

The sender (the publisher) of messages/events is unaware of which microservices would consume them and at which point in time they would do it.

On the receiving end (the subscriber) of this pattern, the microservice expresses an interest in the information shared as messages/events by subscribing to a specific set of information types—or topics, to use a better term. (Note that it is not forced to consume the complete stream of messages; it will be able to distill the relevant information from the rest.)

With a similar perspective, a subscriber is also not aware of the location and status of the publisher, as we can see in the following diagram:

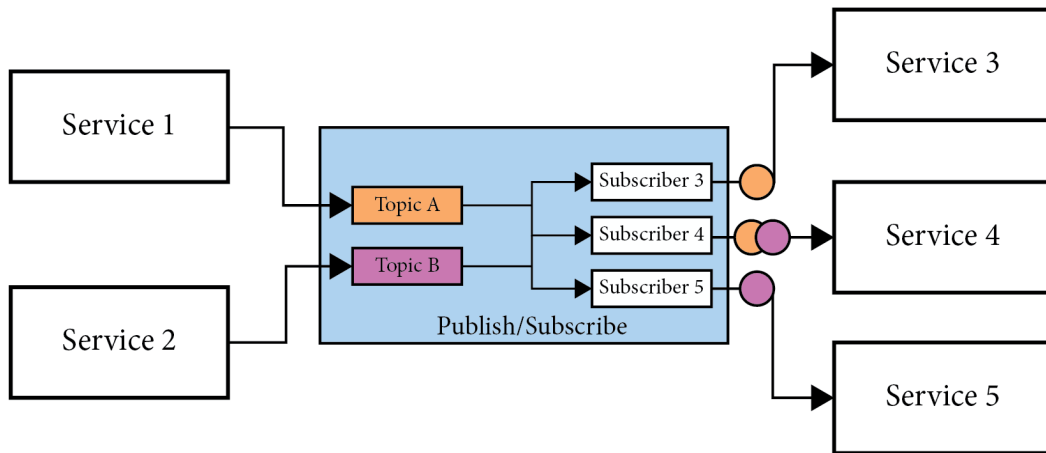


Figure 6.1 – Pub/sub pattern

In *Figure 6.1*, you can see the pub/sub pattern in action: services publish messages specifying the topic—this could be of interest to other services that subscribe to one or many topics.

Over time, more subscribers can be added, with no impact on the publisher.

The pub/sub pattern is often pitted against the service-to-service invocation approach; the first offers a separation in location and time between the request and its processing, while the latter offers a predictable interaction and expectation of response time.

With pub/sub, it's more complex to implement close loops of requests/responses as it's not meant for this style of interaction; on the contrary, the pattern enables each microservice to operate at its own pace, scaling independently and evolving autonomously.

In a service-to-service interaction, the microservices are well aware of each other as they share an agreement (or contract) on the **Application Programming Interface (API)** surface, and they should aim to sustain the pace of the fastest component. Being so closely coupled brings the benefit of simpler interactions with shorter request/response cycles, as there are no intermediaries.

In a discussion of the pub/sub pattern, it might be useful to disambiguate between the concepts of message and event. They both define a piece of information shared between systems, but a message is usually used to pass a request to continue processing elsewhere, while an event brings the news that something significant occurred. An event-driven architecture is centered on the consumption (and emission) of events to which a microservice reacts.

Let's use the example common to all our samples so far—that is, our cookie-selling e-commerce site—as follows:

- An example of an event could be the signaling of a customer accessing our e-commerce site or the expiration time of their shopping cart
- An example of a message could be a request to compensate an activity; for example, to void the impact of an order sent from one microservice to another, previously performed via direct service invocation, asynchronously but as soon as possible

Apart from the conceptual differences and use cases, both messages and events are equally supported by the pub/sub building block of Dapr.

An important role in a pub/sub pattern is played by the messaging system, responsible for the ingestion, safe storage, consumption of messages, and interacting with publishers and subscribers over standard protocols.

The externalization of responsibilities of a messaging system—be it a hosted software package or a messaging system as a service, as commonly provided by most major cloud vendors—is a common pattern of microservice architectures.

The full list of Dapr components supporting the pub/sub brokers can be found at <https://docs.dapr.io/reference/components-reference/supported-pubsub/>. The following is a subset of the supported messaging systems:

- Azure Service Bus (ASB)
- Azure Event Hubs
- **Neural Autonomic Transport System (NATS)**
- Kafka
- **Google Cloud Platform (GCP) Pub/Sub**
- **Message Queuing Telemetry Transport (MQTT)**
- RabbitMQ
- Redis streams

By default, Dapr enables Redis as a pub/sub component, also acting as a state store, in the self-hosted hosting mode.

As a pluggable runtime, Dapr enables the configuration of multiple pub/sub components to be used by the same application.

A Dapr application or client can interact with the pub/sub component via the Dapr API. A simple `POST` request to the Dapr local endpoint exposed by the sidecar with the `http://localhost:<daprPort>/v1.0/publish/<pubsubname>/<topic>` structure would publish data to other Dapr applications subscribing to the same topic named `<topic>` via the `<pubsubname>` configured component.

In addition to the Dapr API, the Dapr **.NET Software Development Kit (SDK)** simplifies the publishing of messages by abstracting the interaction with the API as well as subscribing to topics and receiving messages.

The delivery of messages to subscribing Dapr applications is guaranteed *at least once*; the runtime takes care of all the complexities and specifics of the messaging system, with no need for additional libraries, and guarantees that the message is going to be delivered at least once to any of the Dapr application's instances, positively replying to the Dapr API with a **HyperText Transfer Protocol (HTTP)** result code of 200 or without raising exceptions.

The *at-least-once* delivery mechanism in Dapr enables your application to have competing consumers; messages relevant to the subscribed topics will be split among all the running instances of that specific Dapr application. On the other hand, be aware that if your code needs to make sure that a message will be processed only once, it is your code's responsibility to avoid duplicates.

It is also important to remind you of the resiliency feature in Dapr. As we learned in the *Resiliency* section in *Chapter 4, Service-to-Service Invocation*, we can apply timeout, retry, and circuit breaker policies to a pub/sub component. This capability can be particularly interesting to handle transient errors both in publishing and subscribing to messages. For more information on resiliency with the publish and subscribe building block, please check the Dapr documentation at <https://docs.dapr.io/operations/resiliency/targets/#pubsub>.

As we have just learned the basic concepts of messaging in Dapr with pub/sub, we are now prepared to apply these using ASB as a cloud message broker with Dapr in the next section.

## Using Azure Service Bus (ASB) in Dapr

To introduce the pub/sub building block of Dapr with the ASB implementation, we will develop, in C#, a prototype of a collaboration between some .NET microservices.

The following figure shows what we would like to accomplish:

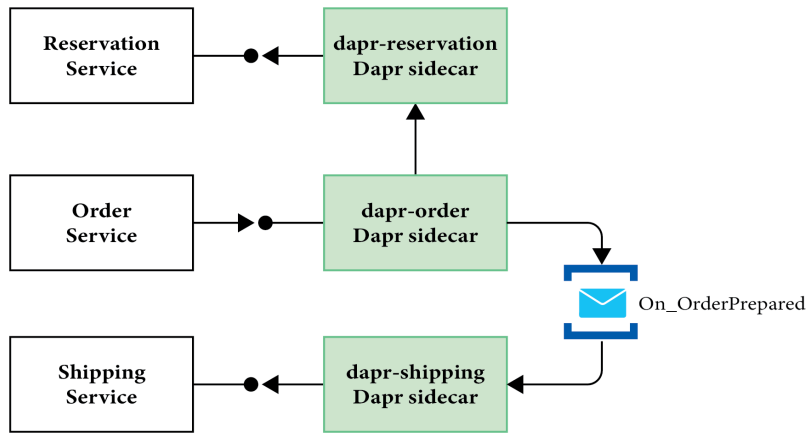


Figure 6.2 – Pub/sub in Dapr with ASB

In *Figure 6.2*, we have the Order service interacting with the Shipping service via pub/sub; this is the portion we are going to develop.

In the `chapter06` folder, you will find many projects. For now, we will focus on `order-service` (`sample.microservice.order`) and `shipping-service` (`sample.microservice.shipping`).

#### Important note: projects and solution

We created samples with several C# projects grouped in a solution, with separate projects for **Data Transfer Objects (DTOs)**, a common library for constants, and so on, referenced by both the service and client projects.

Our intention was to organize the assets in a way that would be easy to consume for you. Two microservices could also agree on the **JavaScript Object Notation (JSON)** format exchanged between the parties and avoid DTOs and references.

To implement the pub/sub pattern, we are going to perform the following steps:

1. Subscribing to a topic
2. Configuring a pub/sub component
3. Publishing to a topic
4. Inspecting the messages

The first step to building our code is to instrument a Dapr service to subscribe to a pub/sub topic.

## Subscribing to a topic

In Dapr, there are two ways for an application to subscribe to topics: **declaratively** and **programmatically**.

With the declarative approach, a `.yaml` formatted file must be composed to inform Dapr of the topic the sidecar has to subscribe to, the application route to invoke, and the pub/sub component to use. The declarative approach enables your application to subscribe to topics without code changes and added dependencies. See the Dapr documentation on **declarative subscriptions** to learn more at <https://docs.dapr.io/developing-applications/building-blocks/pubsub/howto-publish-subscribe/#declarative-subscriptions>.

We are going to explore how to subscribe to topics programmatically, leveraging the abstraction provided by the Dapr SDK for .NET.

We created a `sample.microservice.shipping` project and then referenced the DTO project via the dotnet **Command-Line Interface (CLI)** as follows:

```
PS C:\Repos\practical-dapr\chapter06> dotnet new classlib -o
"sample.microservice.dto.shipping"
PS C:\Repos\practical-dapr\chapter06> dotnet new webapi -o
"sample.microservice.shipping"
```

Let's first examine the `Program.cs` configuration of this ASP.NET project code, shown in the following code snippet—it will have some differences to support pub/sub:

```
var builder = WebApplication.CreateBuilder(args);
var jsonOpt = new JsonSerializerOptions()
{
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
    PropertyNameCaseInsensitive = true,
};
// Add services to the container.
builder.Services.AddControllers().AddDapr(opt => opt.
    UseJsonSerializationOptions(jsonOpt));
... omitted ...
app.UseCloudEvents();
app.MapControllers();
app.MapSubscribeHandler();
app.Run();
```

The changes to the minimal hosting configuration in `Program.cs` are mainly three, outlined as follows:

- With the `.AddDapr()` command, we enable Dapr integration to the ASP.NET MVC.
- The call to `app.MapSubscribeHandler()` registers an endpoint that the Dapr runtime in the sidecar will invoke to become aware of the topics our microservice subscribes to. Without this statement, we would not be able to receive messages from Dapr.
- We use `app.UseCloudEvents` to support the `CloudEvents` message format, as Dapr adheres to the `CloudEvents 1.0`, the specification of which you can find more details at <https://github.com/cloudevents/spec/tree/v1.0>.

We will now move on to decorating our ASP.NET controller to subscribe to a Dapr pub/sub topic.

The following is the method signature of `ShippingController.cs` in the `sample.microservice.shipping` project:

```
namespace sample.microservice.shipping.Controllers;
[ApiController]
public class ShippingController : ControllerBase
{
    public const string StoreName = "shippingstore";
    public const string PubSub = "commonpubsub";

    [Topic(PubSub, Topics.OrderPreparedTopicName)]
    [HttpPost(Topics.OrderPreparedTopicName)]
    public async Task<ActionResult<Guid>> ship(Shipment
        orderShipment, [FromServices] DaprClient daprClient)
    {
        var state = await daprClient.GetStateEntryAsync
            <ShippingState>(StoreName, orderShipment.
                OrderId.ToString());
        state.Value ??= new ShippingState() {OrderId =
            orderShipment.OrderId, ShipmentId =
                Guid.NewGuid() };
        await state.SaveAsync();
        // return shipment Id
        var result = state.Value.ShipmentId;
        Console.WriteLine($"Shipment of orderId {orderShipment.
            OrderId} completed with id {result}");
        return this.Ok();
    }
}
```

```
}
}
```

By decorating the method with the `[Topic(<pubsub component>, <topic name>)]` attribute, we instructed Dapr to subscribe to a topic in the pub/sub component and to invoke this method if a message arrives.

Before we can publish any messages, we need to configure Dapr.

## Configuring a pub/sub component

Our objective is to configure a Dapr component to support the pub/sub building block.

First, we need an external messaging system. We could use the Redis stream made available by the Dapr setup, but instead we are going to create an ASB namespace and leverage it in Dapr for some good reasons (and a strong personal preference of ours).

At this stage in developing our sample, we are running in self-hosted mode. Starting from *Chapter 9, Deploying to Kubernetes*, we will use Kubernetes mode in Dapr. We think that reducing the breadth of changes to the Dapr mode while keeping the components constant helps focus on what is relevant. In addition, the effort of keeping Redis as a reliable messaging store in Kubernetes is significant and beyond the scope of this book.

You can find more information on ASB concepts in the documentation at <https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview>. You can also find detailed instructions on how to create an ASB namespace at <https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-create-namespace-portal>, including how to obtain the connection string we will use later in the chapter.

As developers (or operators), we do not have the responsibility of provisioning ASB topics and subscriptions, nor providing the deployment scripts. Dapr will take care of dynamically creating topics that applications will publish to and their corresponding subscriptions.

In the `\components` folder, we will create a `pubsub.yaml` file with the following structure:

```
apiVersion: daprio/v1alpha1
kind: Component
metadata:
  name: commonpubsub
  namespace: default
```

```
spec:
  type: pubsub.azure.servicebus
  version: v1
  metadata:
    - name: connectionString
      value: # replace
```

Each Dapr component has its metadata configuration; for the `pubsub.azure.servicebus` component type, you can find more details at <https://docs.dapr.io/developing-applications/building-blocks/pubsub/howto-publish-subscribe/>.

How can we know that our service correctly registered the subscription to the topic? By launching `shipping-service`, we can inspect the Dapr output as follows:

```
PS C:\Repos\practical-dapr\chapter06> dapr run --app-id
"shipping-service" --app-port "5005" --dapr-grpc-port "50050"
--dapr-http-port "5050" --components-path "./components" --
dotnet run --project ./sample.microservice.shipping/sample.
microservice.shipping.csproj --urls="http://+:5005"
Starting Dapr with id shipping-service. HTTP Port: 5050. gRPC
Port: 50050
... omitted ...
== DAPR == time="2022-03-26T19:25:40.7256112+02:00" level=info
msg="found component commonpubsub (pubsub.azure.servicebus)"
app_id=shipping-service instance=DB-XYZ scope=dapr.runtime
type=log ver=0.10.0
... omitted ...
Updating metadata for app command: dotnet run --project ./
sample.microservice.shipping/sample.microservice.shipping.
csproj --urls=http://+:5005
You're up and running! Both Dapr and your app logs will appear
here.
```

From the Dapr output, we can see that the `commonpubsub (pubsub.azure.servicebus)` component has been found and set up.

What happened to the ASB messaging system now that we defined a pub/sub component and a subscriber? The following shows what happened:

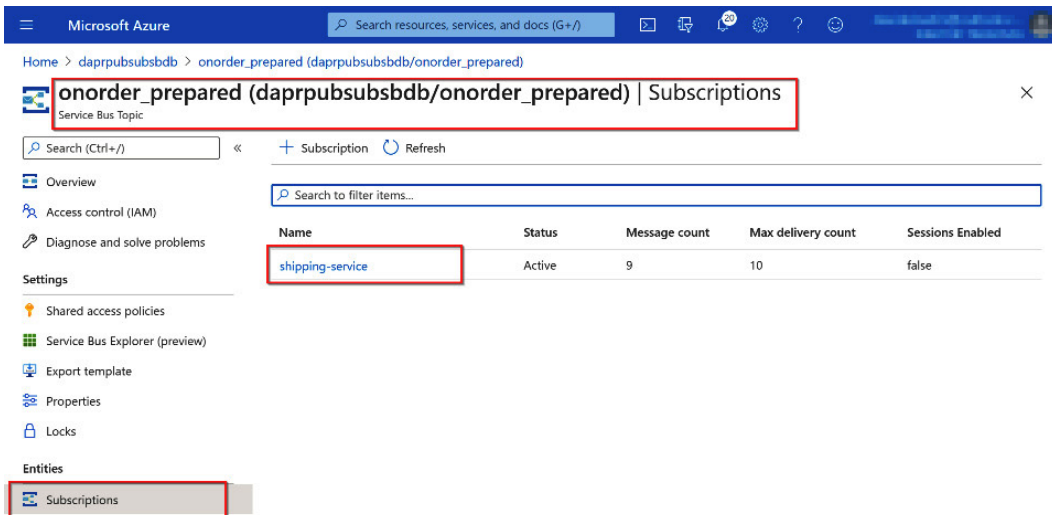


Figure 6.3 – ASB subscriptions of a topic

As you can see in *Figure 6.3*, Dapr created a topic named `onorder_prepared` in the ASB namespace, with a subscription named `shipping-service`.

The next step is to learn how to publish messages to a pub/sub topic in Dapr.

## Publishing to a topic

Any Dapr application can invoke the Dapr runtime reaching the HTTP API, via the SDK or with the Dapr CLI.

We are going to see how the Dapr CLI works as follows:

```
dapr publish --publish-app-id shipping-service --pubsub
commonpubsub -t OnOrder_Prepared -d '{"OrderId": "6271e8f3-
f99f-4e03-98f7-6f136dbb8de8"}'
```

The `dapr publish` command lets us send data to a topic via a pub/sub component.

It's time to verify whether the message has been received.

From the `shipping-service` terminal window, we can see that the message has been received and processed as follows:

```
== APP == Enter shipment start
== APP == Shipment of orderId 6271e8f3-f99f-4e03-98f7-
6f136dbb8de8 completed with id 53c3cc5c-0193-412b-97e9-
f82f3e0d2f80
```

We found evidence that the pub/sub pattern is working as expected from the service output, but how can we inspect the messages?

## Inspecting the messages

By playing around with `dapr publish`, you should soon realize that even the added latency between your development environment and the Azure cloud is very small from a human perspective: each message sent is promptly received by the Dapr microservice.

The ability of a publisher and subscriber to operate independently is one of the major benefits of the pub/sub pattern; we will leverage this to facilitate our learning of Dapr.

While all Dapr applications are running in the local development environment, the messages can be published successfully. If you instead terminate the `shipping-service` Dapr application, there would not be more applications subscribing to the `onorder_prepared` topic.

How can we inspect the messages flowing through our pub/sub component, being ASB in this case? I suggest we manually add a subscription following the instructions at <https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-quickstart-topics-subscriptions-portal#create-subscriptions-to-the-topic>. Let's name the newly created subscription `inspector`, with the default rule that it will receive all the messages sent to the topic.

With the `shipping-service` subscriber application active, we can enqueue some messages by publishing with the Dapr CLI and have the chance to inspect them.

By leveraging the **Service Bus Explorer** feature of ASB via the Azure portal, we can see whether there are pending messages in the **inspector** subscription, as illustrated in the following screenshot:

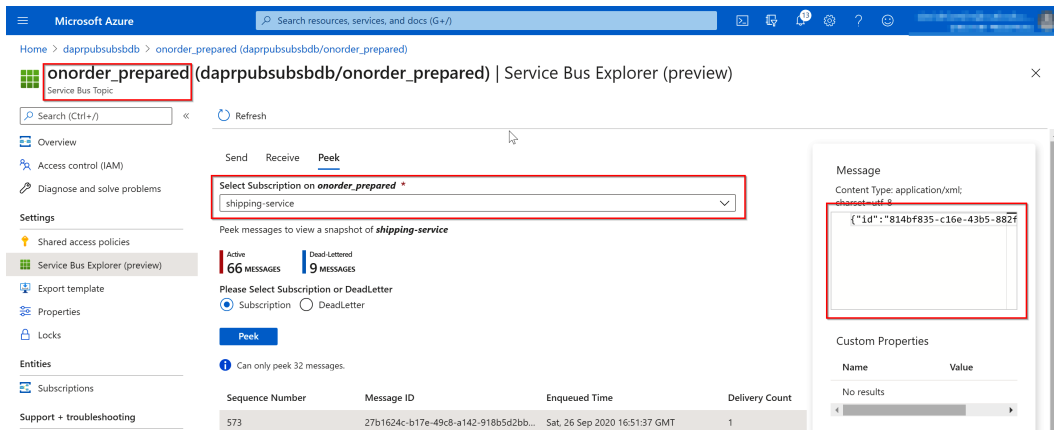


Figure 6.4 – Service Bus Explorer view of a Dapr service subscription

As you can see from *Figure 6.4*, we managed to get several messages in the subscription. With the Peek function, the messages can be inspected without reading them; by selecting one of the messages, we can access the payload.

By inspecting the messages persisted into the auditing subscription, we can see the CloudEvents format being used as illustrated in the following code snippet:

```
{
  "traceid": "00-149c43078b8c752a90ff48e7a0626782-
    ede8db74581c4f7f-01",
  "tracestate": "",
  "data": {
    "OrderId": "6271e8f3-f99f-4e03-98f7-6f136dbb8de8"
  },
  "id": "ce25fa55-c094-4a00-b1f1-0b4ae1a40bcb",
  "datacontenttype": "application/json",
  "type": "com.dapr.event.sent",
  "pubsubname": "commonpubsub",
  "specversion": "1.0",
  "source": "shipping-service",
  "topic": "OnOrder_Prepared"
}
```

The information we sent via the `dapr publish` command is in the `data` field of the message payload, and thanks to the changes we applied in the `Program.cs` file of our ASP.NET project, our code can deal with only the relevant portion of the CloudEvents payload.

**CloudEvents** is a **Cloud Native Computing Foundation (CNCF)** specification for describing event data. It has been adopted by Dapr to format the messages exchanged via pub/sub. See <https://cloudevents.io/> for more information on the specification.

We have had our first experience with the pub/sub component of Dapr. Instead of service-to-service invocation, we interacted with a Dapr application by publishing a message to a topic, relying on an external messaging system such as ASB.

In the next section, we will leverage the Dapr pub/sub building block for a more complex pattern.

## Implementing a saga pattern

As we have learned about how the pub/sub pattern is implemented in Dapr, we can now apply this knowledge to building a more complex scenario, leveraging the saga design pattern for an e-commerce order system.

There are many authoritative sources that discuss saga patterns in detail; we suggest reading <https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga> and <https://vasters.com/archive/Sagas.html>, since it is not in the scope of this book to add anything to an already rich conversation on the subject.

In a microservice architecture, a business activity (for instance, processing an order) could be divided into smaller tasks, carried out by a sequence of microservices interacting with each other.

We have learned that while a microservice gains a lot of benefits by isolating its state, a consequence of this autonomy is that distributed transactions are not conceivable over disparate combinations of databases and libraries. A distributed transaction in this scenario would probably be a bad idea as the transactional context could grow in complexity over time, with more microservices being added or evolving, and have a longer duration; a microservice may not always be available to process the same activity at the same time.

To overcome this reality, a microservice should focus on its scope of responsibility, using local transactions on its own state store/database, publishing a message, or signaling an event to notify the completion of its part of the overall activity. It should also be able to compensate the overall operations with a transaction to reverse the effects on its state, in case the overall sequence of transactions (the saga) is considered failed.

In a nutshell, a saga pattern comes into play in microservice architectures to orchestrate data consistency between the many state stores.

The following diagram shows an example of a saga pattern:

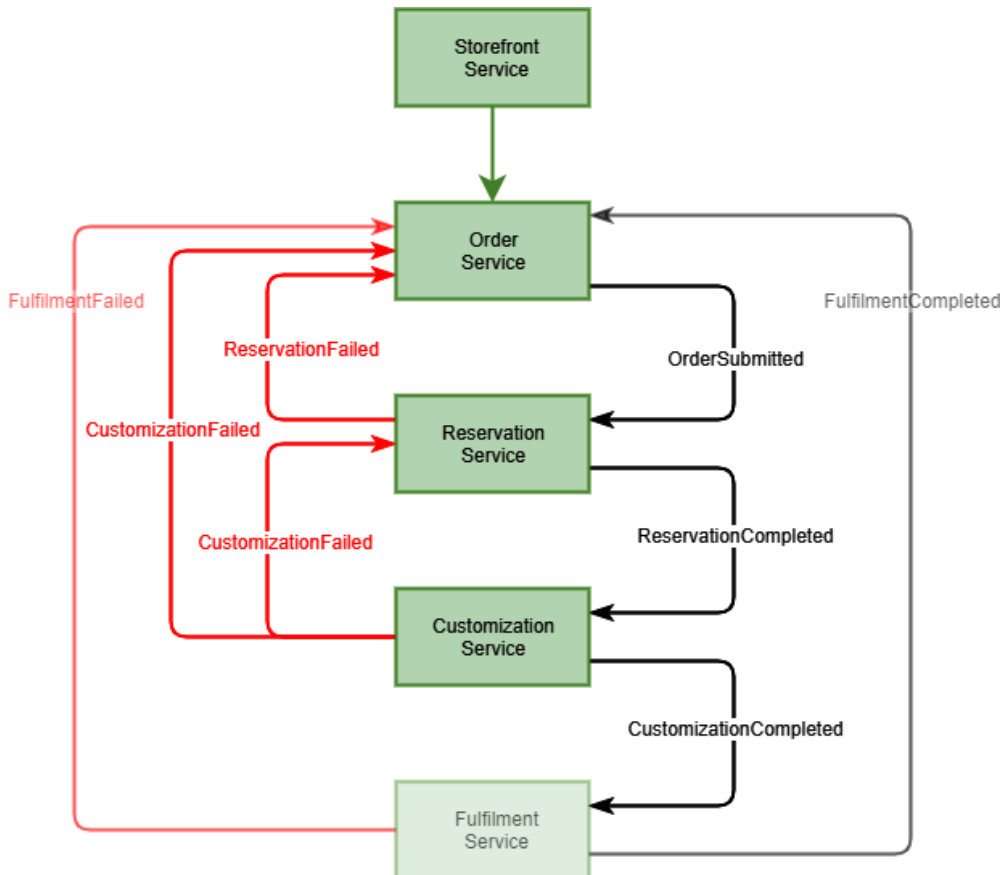


Figure 6.5 – Saga pattern with Dapr

In the context of our cookie-selling e-commerce scenario, the following is how we will structure the saga for our ordering activity, also depicted in *Figure 6.5*:

- An order is submitted (likely from a web frontend) to `order-service`, which registers and publishes it to the `OnOrder_Submitted` topic. `order-service` does not verify the availability of items at the ordered quantity at this stage of the sequence; it is far more likely that if an item landed in the shopping cart, it is available.

`order-service` subscribes to many other topics that signal that, because of irreversible conditions, the order is canceled. As other microservices rely on `order-service` to know the status of an order, the responsibility to keep an order's status up to date lies with `order-service`.

- `reservation-service` subscribes to the `OnOrder_Submitted` topic to learn about new orders and affect the item's available quantities before sending a message to the `OnReservation_Completed` topic. If any of the items' quantities in the order could not be allocated, it compensates for the already allocated items by reversing the effect on the available quantity before publishing the message to the `OnReservation_Failed` topic. By subscribing to the `OnCustomization_failed` topic, `reservation-service` will attempt to compensate the available quantity for the items not already customized, which will be discarded as they cannot be sold to anyone else.
- The other microservice, `customization-service`, operates in a similar fashion by publishing messages to the `OnCustomization_failed` topic to signal the next steps to continue or compensate. It also operates by subscribing to the `OnReservation_Completed` topic, informing the completion of previous steps in the saga.
- As it would not improve this perspective on the saga pattern, for brevity, we will keep the previously created `shipping-service` microservice out of the scope of this sample.

There are three steps to implement the saga, listed as follows:

1. Publishing messages to Dapr
2. Subscribing to a Dapr topic
3. Testing the saga pattern

The first step to build our code is to instrument `order-service` to publish a message to the intended topic.

## Publishing messages to Dapr

After we configure each ASP.NET Web API project to support subscriber handlers and process CloudEvents (as we did in the previous section), we are ready to publish our first message via the Dapr ASP.NET SDK.

Looking at the `SubmitOrder` method in the `OrderController.cs` file of our `sample.microservice.order` project, it's as simple as the following:

```
await daprClient.PublishEventAsync<Order>(PubSub, common.  
Topics.OrderSubmittedTopicName, order);
```

Let's understand the preceding code as follows:

- The preceding code invokes the `PublishEventAsync<T>` method of the instance of type `DaprClient` that our ASP.NET controller gained access to, via **Dependency Injection (DI)**. `DaprClient` is our .NET gateway to all of Dapr's building blocks.

- The first parameter is `pubsubName`. This should match the “commonpubsub” value we specified in the `.yaml` file we placed in the `\components` folder. Our microservice can use multiple pub/sub components at the same time.
- The second parameter is `topicName`, which will influence the routing of our messages and will translate to the ASB topic name itself.
- The third parameter is `data`, the message payload.

**Important note**

This sample code is simple and straightforward; it considers compensating activities and simulates conditions of error in the business logic but does not offer proper exception handling. You should invest your efforts into improving this on your own, as both expected and unexpected conditions should be evaluated to decide whether it makes sense to retry, because of transient errors, or fail and compensate.

As we have just learned how to rely on the Dapr .NET SDK to publish messages to topics, we will subscribe to the appropriate pub/sub topic in the next section.

## Subscribing to a Dapr topic

As we learned in a previous section, *Subscribing to a topic*, an ASP.NET controller method signature can be decorated with the `[Topic]` attribute to subscribe to messages from the topic, as illustrated in the following code snippet:

```
[Topic(PubSub, common.Topics.CustomizationFailedTopicName)]
[HttpPost(common.Topics.CustomizationFailedTopicName)]
public async Task<ActionResult<Guid>>
OnCustomizationFailed(OrderCustomization customization,
[FromServices] DaprClient daprClient)
{
... omitted ...
}
```

To leverage the Dapr ASP.NET integration, the controller’s method should also have a `route` attribute method; this might seem counterintuitive at first, but the Dapr runtime will invoke this method once it receives a message from the topic.

An important point of attention is on the ASP.NET controller’s outcome. As documented in the Dapr documentation at [https://docs.dapr.io/reference/api/pubsub\\_api/#expected-http-response](https://docs.dapr.io/reference/api/pubsub_api/#expected-http-response), the HTTP result does influence how the Dapr runtime handles the message passed to the subscriber. Let’s examine a few possible outputs as follows:

- An HTTP 200 response with an empty payload, or with a "status" key and "SUCCESS" value in a JSON payload, informs the Dapr runtime that the message has been successfully processed by the subscriber.
- An HTTP 200 response with a JSON payload, a "status" key, and a "RETRY" value informs Dapr that the message must be retried; this is helpful if your microservice encounters a transient error.
- Different responses can inform Dapr to log a warning or error, or to drop or retry the message.

All the microservices have been instructed to publish and subscribe to their messages; now, it's time to test the overall scenario.

## Testing the saga pattern

Via the Dapr CLI, as described in the `launch.ps1` file, we can launch the three microservices in the scope of the saga implementation from different terminal windows. Once the Dapr applications have started successfully, we are ready to test the overall saga invoking `order-service` via any of the Dapr sidecars, as you can find in the `order.test.http` file, or as follows:

1. The originating order payload evolved to include special requests for customization is as follows:

```
POST http://localhost:5010/v1.0/invoke/order-service/
method/order
content-type: application/json
{
  "CustomerCode": "Davide",
  "Items": [
    ... omitted ...
  ],
  "SpecialRequests" : [
    {
      "CustomizationId" : "08ffffcc-7591-4702-ffff-
        fff6787bffffe",
      "Scope":
      {
        "ProductCode": "crazycookie",
        "Quantity": 1
      }
    }
  ]
}
```

2. The order is immediately submitted. The `order-service` microservice communicates via pub/sub with the other microservices. The outcome is illustrated in the following output:

```
== APP == Submitted order 17ecdc67-880e-4f34-92cb-
ed13abbd1e68
```

Note that for a Dapr application, or any service in more general terms, to fully leverage the pub/sub pattern in place of a service-to-service interaction, the caller application should not need immediate feedback from the called application before returning a valuable result to its clients. In this case, `order-service` benefits from not being slowed down by the processing time occurring in other Dapr applications.

3. The `reservation-service` microservice allocates the item quantity as follows:

```
== APP == Reservation in 17ecdc67-880e-4f34-92cb-
ed13abbd1e68 of rockiecookie for 4, balance 76
== APP == Reservation in 17ecdc67-880e-4f34-92cb-
ed13abbd1e68 of bussola8 for 7, balance 1
== APP == Reservation in 17ecdc67-880e-4f34-92cb-
ed13abbd1e68 of crazycookie for 2, balance 19
== APP == Reservation in 17ecdc67-880e-4f34-92cb-
ed13abbd1e68 completed
```

4. The `customization-service` microservice is ready to receive special requests for customizing the cookies. Unfortunately, the customization of the *crazycookie* **Stock-Keeping Unit (SKU)** shown in the following code snippet is almost certain to fail:

```
== APP == Customization in 17ecdc67-880e-4f34-92cb-
ed13abbd1e68 of crazycookie for 1 failed
== DAPR == time="2022-09-05T21:36:09.1056547+02:00"
level=error msg="non-retriable error returned from app
while processing pub/sub event ... omitted ...
```

`customization-service` in fact fails, and it publishes a message in `OnCustomization_failed` to notify the saga participants. As you can see, we received an output from the application and from Dapr as well. In this case, the `customization-service` code sample returned a response to inform that, while something unexpected happened, the message should not be retried as the condition of error is considered unrecoverable.

5. `reservation-service` also has the goal to compensate for the failed order customization by releasing the reserved quantities for all items that have not already been customized, and therefore are still sellable.
6. As `reservation-service` subscribes to the `OnCustomizationFailed` topic, it is ready to perform compensating actions as the following output shows:

```
== APP == Reservation in 17ecdc67-880e-4f34-92cb-  
ed13abbd1e68 of rockiecookie for -4, balance 80  
== APP == Reservation in 17ecdc67-880e-4f34-92cb-  
ed13abbd1e68 of bussola8 for -7, balance 8  
== APP == Reservation in 17ecdc67-880e-4f34-92cb-  
ed13abbd1e68 of crazycookie for -1, balance 20  
== APP == Acknowledged customization failed for order  
17ecdc67-880e-4f34-92cb-ed13abbd1e68
```

As an additional note, in the `ReservationController.cs` file, to compensate for the reservation on the item quantities, **Language-Integrated Query (LINQ)** technology has been used to calculate it. As this is not in this book's scope, we encourage you to go read and learn more on the topic from the documentation at <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>.

7. `order-service` subscribes to the `OnCustomizationFailed` topic too. If we look at the ASB namespace in the Azure portal, we should see two subscriptions on the `OnCustomizationFailed` topic.

Also, this microservice receives the following message on the customization failure:

```
== APP == Acknowledged customization failed for order  
17ecdc67-880e-4f34-92cb-ed13abbd1e68
```

With this brief sequence of traces, we were able to implement and appreciate the simplicity and power of the saga pattern with Dapr.

## Summary

In this chapter, we learned how to adopt the pub/sub building block of Dapr to decouple communication between microservices in a way that is far more efficient than we achieved with service-to-service direct invocation, but not without additional effort.

We figured out how to configure the pub/sub component with ASB and how to use the Dapr .NET SDK to instruct ASP.NET controllers to pub/sub messages.

Finally, we discussed the saga design pattern to tackle the complexity of distributed data consistency without resorting to distributed transactions and implemented it in our sample scenario.

In the next chapter, we will explore the resource-binding building block to interact with external services and events.



# Resource Bindings

In this chapter, we will focus on **resource bindings** in **Distributed Application Runtime (Dapr)**, a convenient and pluggable approach to invoking external systems from Dapr microservices and triggering Dapr applications based on external events.

The following are the main topics that we will cover in this chapter:

- Learning how to use Dapr bindings
- Using Twilio output bindings in Dapr
- Ingesting data with the Azure Event Hubs input binding

Learning about Dapr resource bindings is important in the scope of developing new solutions and improving existing ones. While the **publish/subscribe (pub/sub)** pattern we explored in *Chapter 6, Publish and Subscribe*, is helpful in orchestrating asynchronous communication between Dapr applications, the knowledge we get from resource bindings will bring interoperability into our solution.

The very first step in this journey is to learn more about Dapr resource bindings.

## Technical requirements

The code for this chapter can be found on GitHub at <https://github.com/PacktPublishing/Practical-Microservices-with-Dapr-and-.NET-Second-Edition/tree/main/chapter07>.

In this chapter, the working area for scripts and code is expected to be `<repository path>\chapter07\`. In my local environment, it is `C:\Repos\practical-dapr\chapter07`.

Please refer to the *Setting up Dapr* section in *Chapter 1, Introducing Dapr*, for a complete guide to the tools needed to develop with Dapr and work with the samples.

## Learning how to use Dapr bindings

In previous chapters, we devoted most of our attention to the Dapr architecture, and how to use its building blocks to facilitate communication between microservices in the context of the Dapr environment.

With Dapr's service-to-service building blocks, we can directly invoke another microservice as the Dapr runtime takes care of routing requests to their destination and handling retries, among other benefits.

By managing the state, Dapr lifts from our microservice the responsibility of maintaining the plumbing code and the libraries necessary to interact with a persistence layer.

By supporting the pub/sub pattern, Dapr enables microservices to communicate in a loosely coupled fashion and allows our overall architecture to grow in complexity, minimizing the impact on existing portions of code.

All these building blocks focus inward on our microservices, although often, the architecture is not completely isolated as there is a need to reach external systems outside of our microservices, whether it is to receive data, react to external events, or communicate events.

The Dapr bindings building block can be used as an output binding, to invoke an external resource to communicate an event or as an input binding, to trigger our Dapr applications based on an external event.

As an example, these are a few of the currently available output bindings, allowing a Dapr application to execute a specific action on a resource:

- **HyperText Transfer Protocol (HTTP)**
  - Kafka
- **Message Queuing Telemetry Transport (MQTT)**
  - RabbitMQ
  - Twilio
- **Microsoft Azure:** Blob storage, Event Hubs, Cosmos DB, **Azure Service Bus (ASB)**, SignalR, Queue Storage, and Event Grid
- **Amazon Web Services (AWS):** **Simple Storage Service (S3)**, **Simple Queue Service (SQS)**, and **Simple Notification Service (SNS)**
- **Google Cloud Platform (GCP):** Cloud Pub/Sub and Storage Buckets

Some of the available input bindings, allowing a Dapr application to be triggered based on an event raised by the resource, are listed here:

- `cron`
- Kafka
- MQTT

- RabbitMQ
- Azure: Event Hubs, ASB, Queue Storage, and Event Grid
- Amazon: SQ and Kinesis
- GCP: Cloud Pub/Sub

For a complete list of Dapr **input and output (I/O)** bindings, please check the Dapr documentation at <https://docs.dapr.io/reference/components-reference/supported-bindings/>.

To use a binding in Dapr, it must first be configured as a component. Let's see how to configure one of the simplest blocks: a `cron` input binding.

## Configuring a cron input binding

In a local development environment, the `.yaml` files must be located in the `dapr run --app-id "<application>" --components-path "./components"` folder specified in the Dapr **command-line interface (CLI)**. Each Dapr application could have a different path but, as some components are used by several Dapr applications in this book's samples, I will keep all `.yaml` file components in a common folder at the solution level, for simplicity.

A `cron` binding adopts the following configuration:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: cron
  namespace: default
spec:
  type: bindings.cron
  version: v1
  metadata:
    - name: schedule
      value: "@every 10s"
```

More information can be found in the Dapr documentation for bindings at <https://docs.dapr.io/developing-applications/building-blocks/bindings/bindings-overview>.

The relevant settings for the configuration of this component of the `bindings.cron` type are `name` and `schedule`. With an input binding, the configured name will be used by the Dapr runtime to invoke, as POST, the corresponding route at the application endpoint, with the frequency defined in the schedule.

In other input bindings, the trigger will correspond to the arrival of a message or event.

## Testing the cron binding

From an ASP.NET perspective, we need to implement a method with the `cron` route in the ASP.NET controller; this is an example of an addition to the `shipping-service` Dapr project. Here's the code to do this:

```
[HttpPost("cron")]
public async Task<IActionResult> Cron()
{
    Console.WriteLine($"Cron @ {DateTime.Now.ToString()} ");
    return new OkResult();
}
```

As shown in the preceding code snippet, I am not considering the request payload: the intent with the `cron` input binding is just to schedule a recurring request.

By returning a successful result, the ASP.NET controller informs the Dapr runtime that the operation has been completed. It could not be simpler than this.

In the next section, we will learn how to configure and use a more sophisticated output binding.

## Using Twilio output bindings in Dapr

An output binding enables our microservice to actively interact with an external system or service without having to deal with **software development kits (SDKs)**, libraries, or **application programming interfaces (APIs)** other than the Dapr API. In our C# sample, we will use the Dapr .NET SDK to abstract this interaction.

In the previous chapter, *Chapter 6, Publish and Subscribe*, we introduced the `shipping-service` project: this Dapr application subscribes to the `OnOrder_Prepared` topic to be informed once all the steps in the order-preparation saga reach a positive conclusion.

We intend to increase the functionality of this microservice by informing the customer that the order is shipped. To do so, we can leverage a notification service such as **Twilio** to send the customer a **Short Message Service (SMS)** message, as follows:

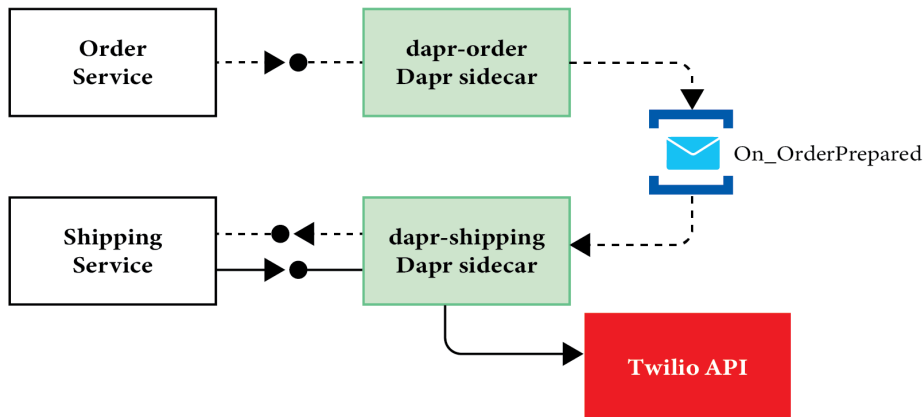


Figure 7.1 – Twilio output binding added to the shipping service

In *Figure 7.1*, you can see the evolution of the `shipping-service` Dapr service: an output resource binding of the Twilio type is being adopted.

We will start working on the binding by following the following four simple steps:

1. Signing up for a Twilio trial
2. Configuring a Twilio output binding
3. Signaling via the output binding
4. Verifying the notification

We will begin with the first step and sign up for a Twilio trial.

## Signing up for a Twilio trial

The first step is to sign up for a Twilio trial. You can request a Twilio free trial at <https://www.twilio.com/>. As we are going to send a text message to the customer in our sample, you will need to register a valid phone number. I registered my own mobile number for this purpose.

### Important note

Be aware of the Twilio limitations for a free trial, which you can find here: <https://support.twilio.com/hc/en-us/articles/360036052753-Twilio-Free-Trial-Limitations>. As an example, you can send text messages only to a validated phone number, and the message will start with some default text.

Once you have an account and an active project, there are two strings you need to collect from the Twilio page, as illustrated in the following screenshot:

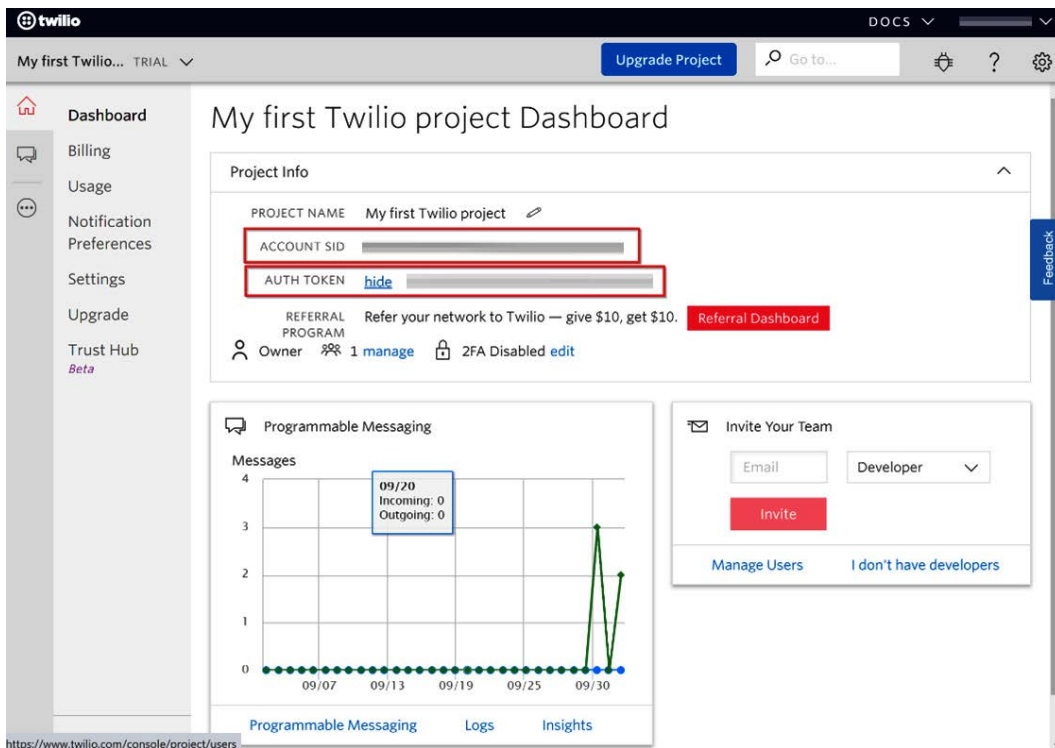


Figure 7.2 – Twilio account security identifier (SID) and authentication (auth) token

**ACCOUNT SID** and **AUTH TOKEN** from *Figure 7.2* are the configuration strings that will be used to configure the Twilio binding in Dapr.

## Configuring a Twilio output binding

The Twilio binding specification details are available in the Dapr documentation repository at <https://docs.dapr.io/operations/components/setup-bindings/supported-bindings/twilio/>; we have to create a configuration file to access Twilio with our account and credentials. Here is the content of the `components\twilio.yaml` file:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: twilio
spec:
```

```
type: bindings.twilio.sms
version: v1
metadata:
- name: fromNumber # required.
  value: <omitted>
- name: accountSid # required.
  value: <omitted>
- name: authToken # required.
  value: <omitted>
```

Examining the previous configuration file, in the output binding of type `bindings.twilio.sms`, I did not specify the `toNumber` metadata key; this will be influenced by our code.

The `accountSID` and `authToken` keys must be set to the values we gathered from the Twilio web portal.

Next, we will let the application know when a text message should be sent.

## Signaling via the output binding

We have to launch our Dapr application and trigger the subscription by sending a test message.

We can use the Dapr CLI to launch `shipping-service`, as follows:

```
dapr run --app-id "shipping-service" --app-port "5005" --dapr-
grpc-port "50050" --dapr-http-port "5050" --components-
path "./components" -- dotnet run --project ./sample.
microservice.shipping/sample.microservice.shipping.csproj
--urls=http://+:5005
```

In the Dapr output logs, we should see an acknowledgment of the binding sent by the runtime, as follows:

```
== DAPR == ... level=info msg="found component twilio (bindings.
twilio.sms)" app_id
=shipping-service instance=DB-XYZ scope=dapr.runtime type=log
ver=1.8.0
```

The previous Dapr log trace should be followed by a similar output, which we can see here:

```
== DAPR == ... level=info msg="successful init for output
binding twilio (bindings.twilio.sms)" app_id=shipping-service
instance=DB-XYZ scope=dapr.runtime type=log ver=1.8.0
```

Let's explore how we can use the output binding in the C# code of our ASP.NET ShippingController.cs controller file, as follows:

```
[Topic(PubSub, Topics.OrderPreparedTopicName)]
[HttpPost(Topics.OrderPreparedTopicName)]
public async Task<ActionResult<Guid>> ship(Shipment
orderShipment, [FromServices] DaprClient daprClient)
{
    var state = await daprClient.GetStateEntryAsync
        <ShippingState>(StoreName, orderShipment.OrderId.
ToString());
    state.Value ??= new ShippingState() {OrderId =
    orderShipment.OrderId, ShipmentId = Guid.NewGuid() };
    await state.SaveAsync();
    // return shipment Id
    var result = state.Value.ShipmentId;
    Console.WriteLine($"Shipment of orderId
    {orderShipment.OrderId} completed with id {result}");
    var metadata = new Dictionary<string, string>();
    metadata.Add("toNumber", "<omitted>");
    await daprClient.InvokeBindingAsync<string>
    ("twilio", "create", $"Dear customer, your order with
    {orderShipment.OrderId} completed and shipped",
    metadata);
    Console.WriteLine($"Shipment of orderId
    {orderShipment.OrderId} notified to customer");
    return result;
}
```

The instance of `DaprClient` we run in the ASP.NET controller gives us access to the `daprClient.InvokeBindingAsync` method. The `metadata` parameter is a key-value dictionary that can be used to influence the metadata configured in `component.yaml`: if you remember, we did not specify the `toNumber` key as it is the microservice's responsibility to gather it from the order (or from another microservice managing the customer data).

The first and second parameters specify the `twilio` name of the configured binding and the intended `create` operation, among those supported by the binding.

We can simulate a message via the Dapr CLI with the `dapr publish` command, as follows:

```
dapr publish --pubsub commonpubsub -t OnOrder_Prepared -d
'"{\"OrderId\": \"08ec11cc-7591-4702-bb4d-7e86787b64fe\"}"'
```

From the `shipping-service` output, we see the message has been received and the *shipping* has been completed.

All went fine with our code, as the Dapr runtime responded positively to our request. We just need to notify the customer of this, as the last step.

## Verifying the notification

The Dapr output binding allows us to interact with an external system. Aside from positive feedback (no exceptions) from the Dapr API, there is only one other thing we can do to verify the process completion: check our phone for text messages! The following screenshot shows this being done:

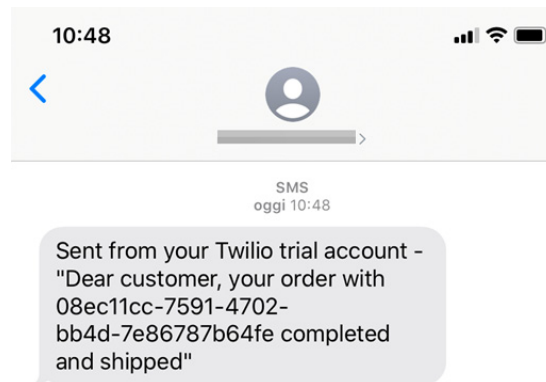


Figure 7.3 – Twilio SMS received

As you see from *Figure 7.3*, we received the notification sent from `shipping-service` via Twilio.

In this sample, we triggered a Dapr application via pub/sub, which in turn signaled an event via the output binding. We could also leverage the runtime endpoint to test it directly, as follows:

```
POST http://localhost:5050/v1.0/bindings/twilio HTTP/1.1
content-type: application/json
{
  "data": "cookies ready from Dapr",
  "metadata": {
    "toNumber": "<omitted>"
  }
}
```

```

    },
    "operation": "create"
  }

```

Consistent with the previous use of the .NET SDK, `toNumber` is set as a value in metadata, and `operation` is set to `create`.

We have completed our first sample with the Dapr output binding. In the next section, we will understand how to use the Dapr binding to trigger our microservices.

## Ingesting data with the Azure Event Hubs input binding

In a previous section of this chapter, we learned how to implement a simple input binding thanks to the `cron` sample. In this section, we will explore another input binding, leveraging the Azure Event Hubs cloud messaging service, by implementing it in the context of `reservation-service`.

The responsibility of `reservation-service` is to allocate quantities of a certain product (cookies) as a new order comes in. In this context, we never considered that if there is a process to reserve (and therefore subtract) quantities, then there should be an equivalent—but opposite—process to increment the available quantity. This is our chance to fix the business logic of our sample.

In the context of our sample's cookie-selling e-commerce site, let's suppose there is an external service overseeing the manufacturing process, which produces cookies to be sold and/or customized according to customers' requests, depending on forecasts and short-term incoming orders. This manufacturing service is not going to participate with other microservices via Dapr: the only link between the two subsystems is via a stream of events through an Azure Event Hubs channel, as illustrated in the following diagram:

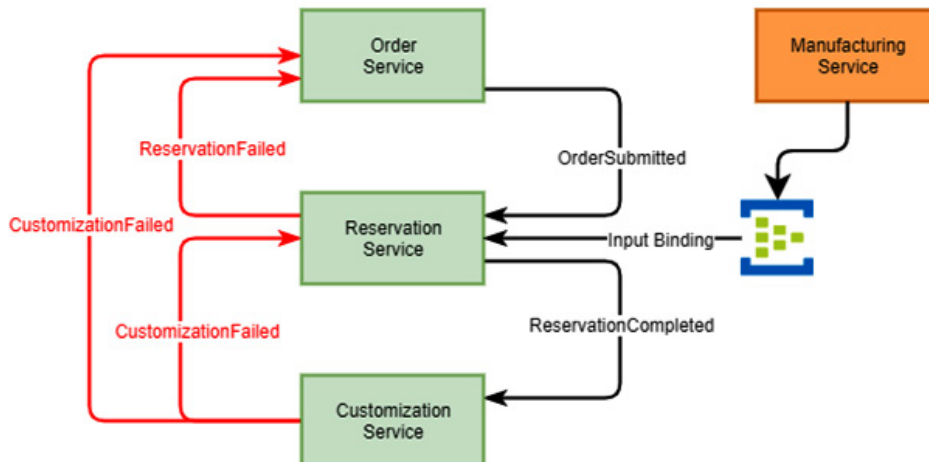


Figure 7.4 – Manufacturing service interaction with reservation

As seen in *Figure 7.4*, the overall context of the saga, influencing the communication pattern for the requests and compensating the transactions, will be affected by additional data coming from an external subsystem, via a Dapr input binding, making it all the more important to orchestrate messaging.

This is what we will implement in the following sections.

## Creating an Azure Event Hubs binding

In order to configure an Azure Event Hubs input binding, we first have to provision it in Azure.

First, we create an Azure Event Hubs namespace and an event hub. Detailed step-by-step instructions on how to provision these resources on Azure can be found at <https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-create>.

The following screenshot from the Azure portal shows the result:

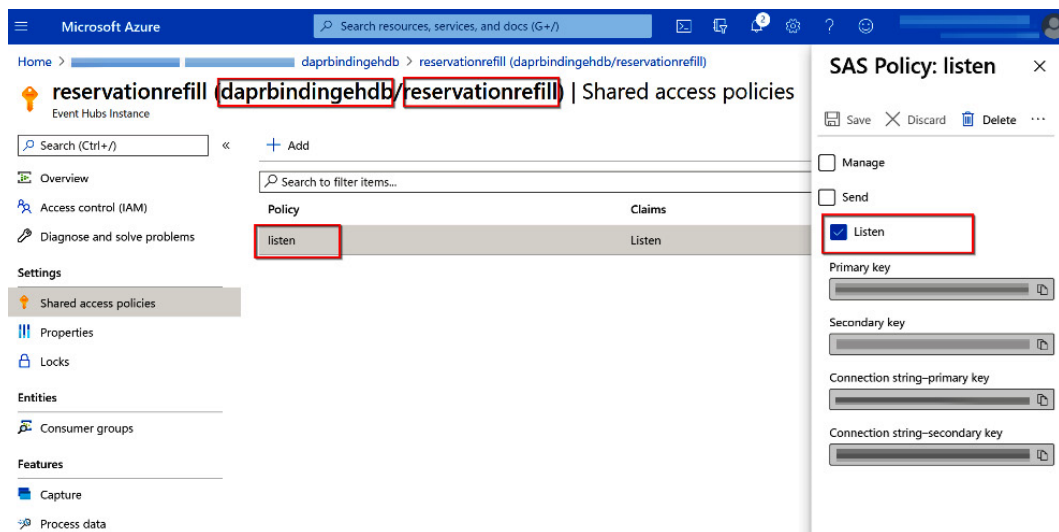


Figure 7.5 – Azure Event Hubs policy

As you can see in *Figure 7.5*, I created an Azure Event Hubs `daprbindingehdb` namespace and a `reservationrefill` event hub, and I configured an access policy with **Listen** claims.

The access policy with **Listen** rights should be enough for an input binding; it would have needed **Send** rights to be used with an output binding instead.

Finally, we need to create an Azure storage account; this resource will be used by the Dapr input binding implementation for Event Hubs to keep track of the offset, the point reached in reading events. Please refer to the step-by-step instructions at <https://docs.microsoft.com/en-us/azure/storage/common/storage-account-create?tabs=azure-portal>.

In the following screenshot, you can see that a storage account has been created from the portal:

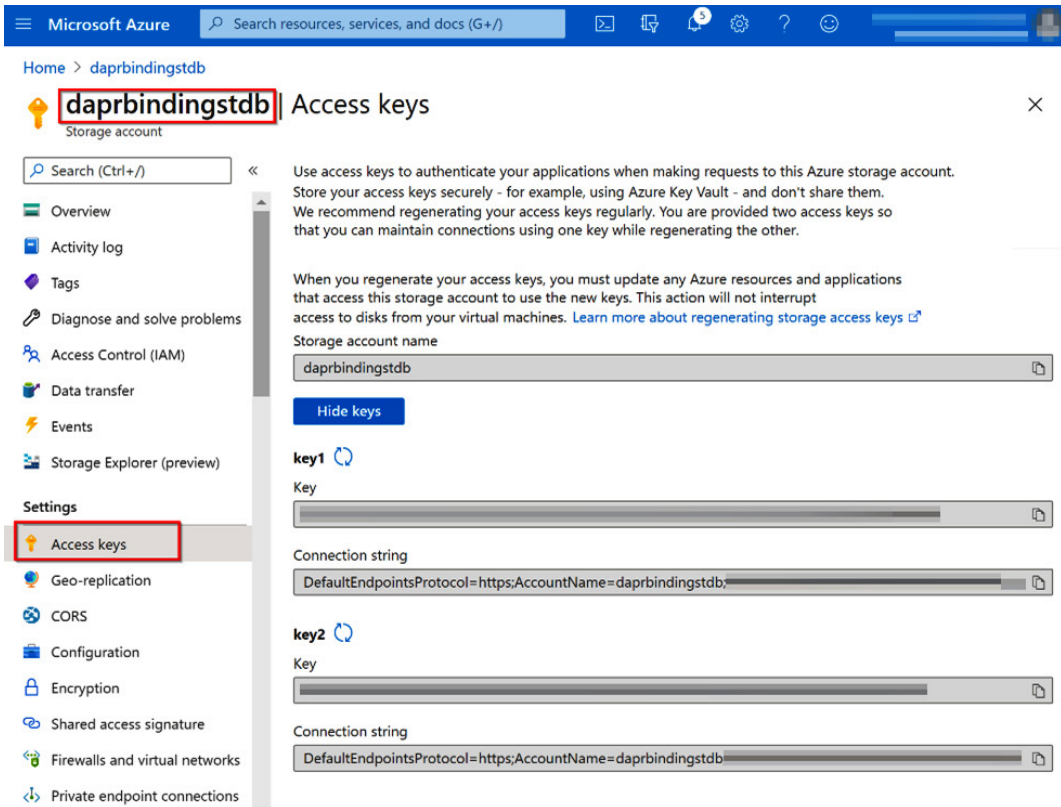


Figure 7.6 – Storage account for the Event Hubs processor host

In *Figure 7.6*, you can see that I created an Azure storage account and obtained the connection string. This information will be used in the next steps to configure the Dapr component.

## Configuring the input binding

Considering the resources that we previously created, the following file in `components\binding-eh.yaml` is needed to instruct Dapr to activate an input binding:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: reservationinput
  namespace: default
```

```
spec:
  type: bindings.azure.eventhubs
  version: v1
  metadata:
    - name: connectionString      # Azure EventHubs connection
      string
      value: "<omitted>"
    - name: consumerGroup        # EventHubs consumer group
      value: "group1"
    - name: storageAccountName   # Azure Storage Account Name
      value: "<omitted>"
    - name: storageAccountKey    # Azure Storage Account Key
      value: "<omitted>"
    - name: storageContainerName # Azure Storage Container Name
      value: "inputbinding"
```

The Azure Event Hubs specification as a binding is available at <https://docs.dapr.io/operations/components/setup-bindings/supported-bindings/eventhubs/>.

As you can note from the preceding configuration, an Azure Storage account is also required to persist Event Hubs checkpoint data.

Dapr is now ready to receive messages (events, in this case). Let's focus on the input binding.

## Implementing an Azure Event Hubs input binding

In `reservation-service`, we implement a new method to receive events via the input binding.

As we learned in previous sections, the name of the ASP.NET route must match the name configured in the Dapr component. In the following code snippet from the `sample.microservice.reservation\Controllers\ReservationController.cs` file, you can see that the attribute reflects the same `reservationinput` component name:

```
[HttpPost("reservationinput")]
public async Task<IActionResult> Refill([FromServices]
    DaprClient daprClient)
{
    using (var reader = new
        System.IO.StreamReader(Request.Body))
    {
```

```
        ... omitted ...
        var stateItem = await daprClient.
            GetStateEntryAsync<ItemState>(StoreName_item,
            SKU);
        stateItem.Value ??= new ItemState() { SKU =
            SKU, Changes = new List<ItemReservation>() };
        stateItem.Value.BalanceQuantity += Quantity;
        await stateItem.SaveAsync();
        ... omitted ...
    }
    return new OkResult();
}
```

The method signature we use in this case is slightly different from that in the previous sample. We need to interact with the Dapr infrastructure to gather our microservice state; therefore, `daprClient` is now a parameter.

In the preceding code, we are making many assumptions, such as the payload of the messages being **JavaScript Object Notation (JSON)** and with a specific schema, to keep this exercise simple.

Specific to Dapr, as the event we receive is intended to contain information for a single item, we retrieve the state via the .NET SDK, update the balance quantity, and subsequently save it back to the state store.

Our Dapr `reservation-service` application is ready to receive and process events.

## Producing events

Using the Azure Event Hubs documentation, available at <https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/eventhub/Azure.Messaging.EventHubs/README.md#publish-events-to-an-event-hub>, as a starting point, we can create a C# console project to simulate the output of the manufacturing service: a continuous stream of new cookies coming out of the ovens is signaled via Azure Event Hubs to the Dapr input binding.

You can see the overall code in `Program.cs` of the `Console` project folder. It continuously sends a refill of a random selection of a cookie's **stock-keeping unit (SKU)** to Event Hubs, which in turn is used as an input binding to `reservation-service`.

We can launch the console via the command line, passing `connectionString` as a parameter, as shown in the following snippet:

```
PS C:\Repos\practical-dapr\chapter07> dotnet run --project .\
Console\chapter07.csproj "... omitted ..."
Started sender
Sent batch @ 22:58:51
Sent batch @ 22:58:53
...
```

The event producer starts to send batches of events to the Azure Event Hubs resource.

#### Important note

While working with Azure messaging services such as ASB and Azure Event Hubs, it is highly recommended to install the Azure Service Bus Explorer suite by Paolo Salvatori; you can find out more at <https://github.com/paolosalvatori/ServiceBusExplorer>. Although some of the features offered by this powerful tool have been included in the Azure portal, Service Bus Explorer continues to be the best tool for anyone developing with the Azure messaging stack.

Let's relaunch the `reservation-service` Dapr application so that our newest input binding can be invoked as simply as an ASP.NET call by the Dapr sidecar:

```
PS C:\Repos\practical-dapr\chapter07> dapr run --app-
id "reservation-service" --app-port "5002" --dapr-grpc-
port "50020" --dapr-http-port "5020" --components-path "./
components" -- dotnet run --project ./sample.microservice.
reservation/sample.microservice.reservation.csproj
--urls="http://+:5002"
```

From the `reservation-service` output, we can verify that the input binding is receiving events via Dapr from an external subsystem, as follows:

```
== APP == Refill of crazycookie for quantity 1, new balance 44
== APP == Refill of bussola1 for quantity 1, new balance 160
... omitted ...
```

This step concludes the implementation of the input binding in our microservice.

## Summary

In this chapter, we explored the I/O binding building blocks of Dapr, and we learned how, with Twilio output binding, we can notify customers via text message without having to deal with libraries, SDKs, and the plumbing code, as it all boils down to a simple call to the Dapr runtime.

We then established a communication channel between a microservice of our sample e-commerce architecture with an external subsystem; both are unaware of each other's implementation details, and our microservice is unaware of how to interact with Azure Event Hubs as the messaging bus.

The `reservation-service` application is placed at the center of our sample architecture, unlike the other microservices.

As a point of attention, the sample code doesn't deal with application-level retries, which could be relevant if the strong consistency of state management and an elevated request rate prevent a reservation from always completing nicely. While this condition should be addressed with more solid code, it does help to expose a case of potential stress in the application, which you might want to tackle as a side exercise.

In the next chapter, we will discover how we can tackle this scenario of high-frequency access to several small, independent state and logic units by introducing Dapr actors.

# 8

## Using Actors

In this chapter, you will learn about the powerful virtual actor model, as implemented in Dapr, and how to leverage it in a microservices-style architecture, along with the pros and cons of different approaches. The actor model enables your Dapr application to efficiently respond to scenarios of high resource contention by streamlining state management.

In this chapter, we will cover the following main topics:

- Using actors in Dapr
- Actor concurrency, consistency, and lifetime
- Implementing actors in an e-commerce reservation system

The entry barrier for adopting actors in Dapr is lower than the complexity behind the core concepts of the virtual actor pattern. Nonetheless, a solid understanding of the scenarios for actors – including the ability to recognize bad practices and avoid any pitfalls – is a prerequisite for their adoption. Therefore, we will start by providing an overview of the actor model before moving on to its lab implementation with Dapr.

### Technical requirements

The code for this chapter can be found on GitHub at <https://github.com/PacktPublishing/Practical-Microservices-with-Dapr-and-.NET-Second-Edition/tree/main/chapter08>.

In this chapter, the working area for the scripts and code can be found at `<repository path>\chapter08\`. In my local environment, it can be found here: `C:\Repos\dapr-samples\chapter08`.

Please refer to the *Setting up Dapr* section of *Chapter 1, Introducing Dapr*, for a complete guide on the tools needed to develop with Dapr and work with the examples provided.

## Using actors in Dapr

The actor model in Dapr adopts the concept of **virtual actors**: a simplified approach to a complex combination of design challenges. Virtual actors originate from the Microsoft *Orleans* project – a project that inspired the design of Dapr. If you want to deepen your knowledge of its history, the respective research paper can be found at <https://www.microsoft.com/en-us/research/project/orleans-virtual-actors/>.

In the virtual actor pattern, the state and behavior of a service are tightly intertwined, and the actor's lifetime becomes orchestrated by an external service or runtime. Because of this, the developers are lifted from the responsibility of governing concurrent access to the resource (the virtual actor) and its underlying state.

These concepts will become clearer when we analyze how the virtual actor pattern is implemented in Dapr in the next section.

## Introduction to the virtual actor pattern

In Dapr, the interaction between a client and a service counterpart happens via a direct call, with a *service-to-service invocation*, or indirectly via a message, with a *publisher and a subscriber*, as seen in the previous chapters. The Dapr application acting as the service then accesses the state to read and/or manipulate it.

If we consider the components involved in this class of interactions, it all boils down to the following:

- A client remote call
- A service processing the request
- A database managing the stored information

We have just listed the same number of interactions as seen in a classic three-tier architecture. A microservice architecture affects several aspects of client/service communication by introducing new patterns and capabilities; nevertheless, it can't escape the laws of physics, such as network latency and storage **input/output (I/O)** latency.

As our Dapr application is going to operate in a highly distributed environment (starting from *Chapter 9, Deployment to Kubernetes*), the three aforementioned components could be located in separate nodes or even be part of different services, and therefore traverse the network and/or application boundaries, each of which adds its share of latency.

Caching helps reduce latency, bringing information closer to the service by improving the performance of repeated reads from the source of information: the database. At the same time, it introduces the complexity of managing consistency, which was previously solved by the database: it's difficult to keep a cache relevant while updates are performed. Once you try to solve this consistency issue by

controlling access to cached information, the complexity of concurrency promptly emerges. Caching is a powerful mechanism, but once you try to strengthen it from its consistency and concurrency perspectives, it risks falling short of its declared objective.

The virtual actor pattern that's implemented in Dapr tries to approach this question differently. It could help us understand the actor model if we consider the starting point and its challenges. Let's do this by analyzing the current status of the backend of our cookie-selling *Biscotti Brutti Ma Buoni* e-commerce site.

The following screenshot depicts the interactions between our sample Dapr applications and their state:

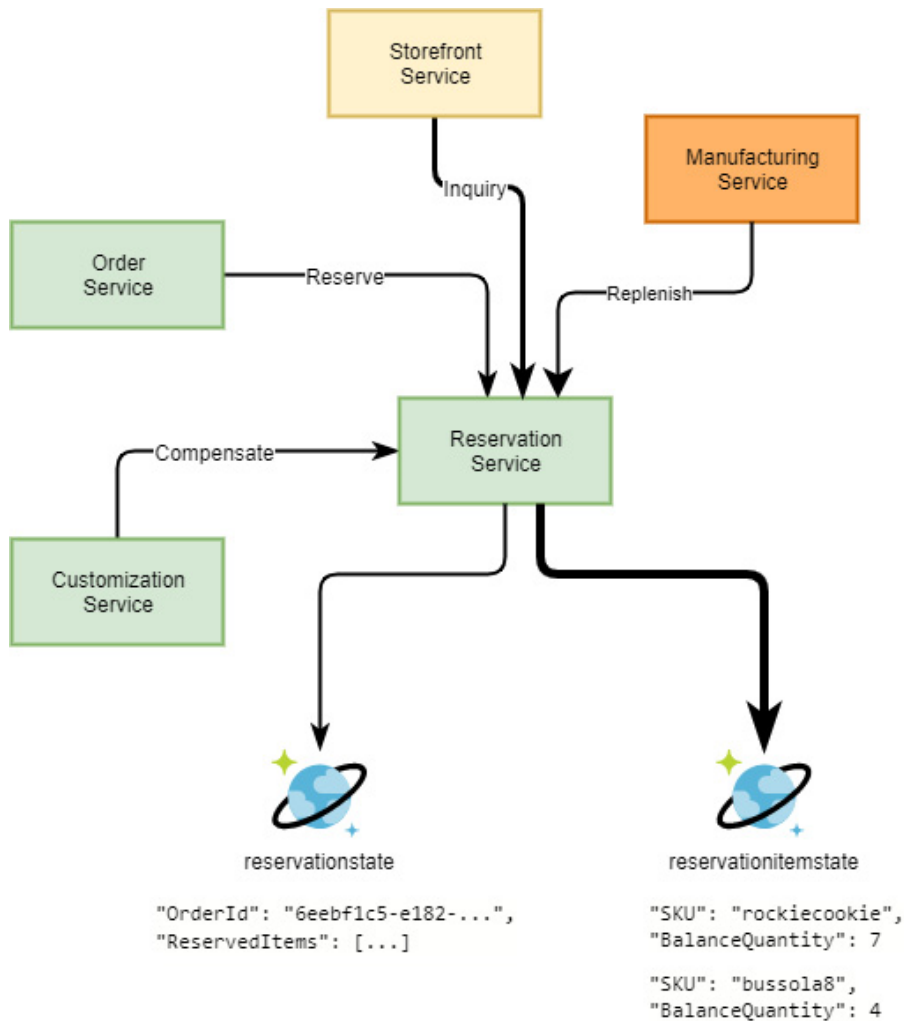


Figure 8.1 – Status quo of the sample architecture

As you can see from *Figure 8.1*, the **Reservation Service** (also known as the `reservation-service` Dapr application) sits at the center of many interactions and participates with other Dapr applications in the saga we explored in *Chapter 6, Publish and Subscribe*, which receives updates from the manufacturing service and responds to requests from our storefront service (which is generic since using a much richer **user interface (UI)** is outside the scope of this book).

The `reservation-service` Dapr application relies on two state stores – `reservationstate` and `reservationitemstate`:

- The `reservationstate` state store keeps track of the products (SKU) that have been reserved by an order. This is useful when we're compensating the order in case of a product customization failure.

We can predict that there will be a limited number of interactions with the state for the specific order of successfully fulfilled ones.

The population of `reservationstate` is always increasing, with operations that concentrate on the state for newly added orders.

- The `reservationitemstate` state store is used differently: it keeps track of the balance of each product's quantity; orders diminish its value, while replenishments increase it. Each order (and its potential compensating activities), storefront request, or manufacturing action equates to state retrieval and updating of the specific item.

We adopt strong concurrency when managing the `reservation-service` state to avoid inconsistent updates. As a side effect, we risk increasing conflicts of updates in the case of sustained growth in requests, leading to more retries to avoid transferring the impact to clients.

The population of `reservationitemstate` becomes stable over time, one for each SKU, with operations being distributed evenly or unevenly, depending on the SKU's popularity.

As shown in the preceding diagram, the same number of data retrieval or manipulation requests are reaching the state store – Azure Cosmos DB, in our example.

By introducing the Dapr actor model, we can create a virtual representation of each `reservation-itemstate` state and bind them with the service code counterpart.

The following diagram shows where the Dapr actor model will be introduced:

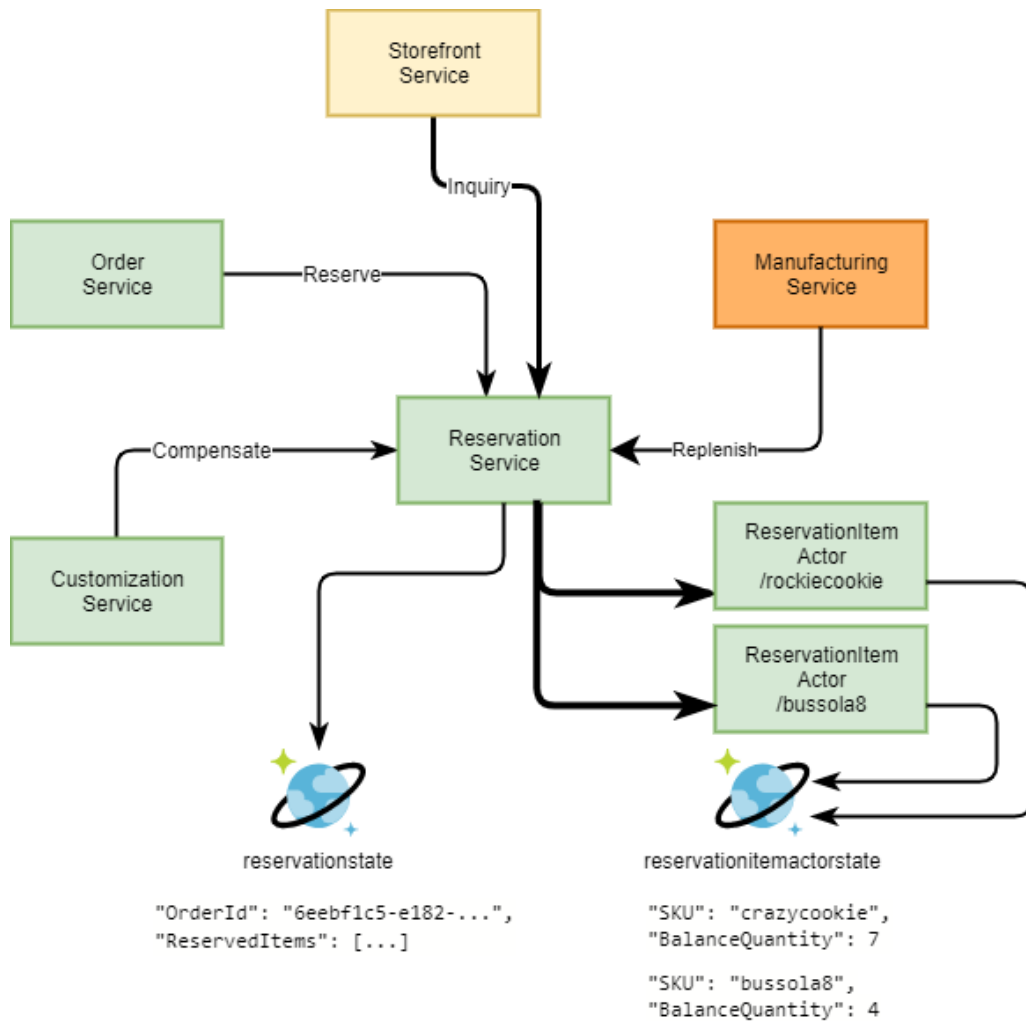


Figure 8.2 – Dapr actor model introduced in the sample architecture

The preceding diagram shows the evolution of our sample architecture: we introduce a new `reservationactor-service` Dapr service that offers access to the population of Dapr actors of the `ReservationItemActor` type.

As we will learn later in this chapter, Dapr manages the lifetime and distribution of actors in the hosting environment (locally or in Kubernetes). We can expect that each of the SKUs typically sold by our e-commerce site will have a corresponding actor instance, ready to receive requests.

What do Dapr (virtual) actors provide to our sample microservice architecture? The business logic that manages the SKU balance quantity, refactored from `reservation-service` to the code of the `ReservationItemActor` actor, will leverage the state that's being maintained in the actor instance itself. This is kept closely in sync by the Dapr runtime.

With actors in Dapr, their state and behavior become intertwined, and the load gets distributed over the hosting platform for resiliency. No additional effort is required from the developer since every aspect is managed by the Dapr runtime and platform.

Accessing a Dapr actor is governed by a turn-based policy: a read operation will find the state information in memory, equally as fast as an item in the cache, while update operations will reach the state store's database without facing the concurrency issues we have discussed so far.

Before we finish this overview, let's learn how to configure a state store for Actors in Dapr.

## Configuring a new state store

We learned how to configure a state store in *Chapter 5, Introducing State Management*. With that knowledge, we will set up a new Azure Cosmos DB state store to keep the state of our actors.

In the same chapter, we also learned that only state stores that support transactions and **ETAg** can be used with actors in Dapr: the Azure Cosmos DB state store is one of these.

For a complete list, see the Dapr documentation at <https://docs.dapr.io/reference/components-reference/supported-state-stores/>.

The following `.yaml` file is an example of a state store configuration:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: reservationitemactorstore
  namespace: default
spec:
  type: state.azure.cosmosdb
  metadata:
    - name: url
      value: ...omitted...
    - name: masterKey
      value: ...omitted...
    - name: database
      value: state
    - name: collection
```

```
value: reservationitemactorstore
- name: actorStateStore
  value: "true"
```

The changes we've made to support the actor model are minimal. We only need to set the `actorStateStore` metadata to `true`.

Now that we have configured a new state store, which I have named `reservationitemactorstore`, we should verify it.

## Verifying the configuration

We can verify that the configuration has been applied by launching any of the existing Dapr applications; it will look for the `.yaml` files in the `components` path.

In the following Dapr application log, only the significant lines have been extracted:

```
== DAPR == time="..." level=warning msg="either no actor state
store or multiple actor state stores are specified in the
configuration, actor stores specified: 0" app_id=order-service
instance=DB-XYZ scope=dapr.runtime type=log ver=1.8.0
== DAPR == time="..." level=info msg="component loaded. name:
customizationstore, type: state.azure.cosmosdb" app_id=order-
service instance=DB-XYZ scope=dapr.runtime type=log ver=1.8.0
```

In the previous output, we can see that the message "either no actor state store or multiple actor state stores are specified" is presented when a state store is found that has not been configured as an actor state store:

```
== DAPR == time="..." level=info msg="component loaded. name:
reservationitemactorstore, type: state.azure.cosmosdb" app_
id=order-service instance=DB-XYZ scope=dapr.runtime type=log
ver=1.8.0
== DAPR == time="..." level=info msg="actor runtime started.
actor idle timeout: 1h0m0s. actor scan interval: 30s" app_
id=order-service instance=DB-XYZ scope=dapr.runtime.actor
type=log ver=1.8.0
== DAPR == time="..." level=info msg="starting connection attempt
to placement service: localhost:6050" app_id=order-service
instance=DB-XYZ scope=dapr.runtime.actor type=log ver=1.8.0
...
```

```
== DAPR == time="..." level=info msg="established connection  
to placement service at localhost:6050" app_id=order-service  
instance=DB-XYZ scope=dapr.runtime.actor type=log ver=1.8.0
```

Since it has not been preceded by any warning message, the `reservationitemactorstore` component has been recognized as a state store for actors. If you incorrectly configure two state stores as `actorStateStore`, you will receive the same warning we saw previously since only one is permitted. No warning is a good sign. In the final output message, we received confirmation that a connection to the Dapr placement service has been established.

This extensive overview gave us a better understanding of how actors in Dapr work to help us build scalable applications. Before we move on to the implementation, we need to understand a few more concepts, including the actor's lifetime.

## Actor concurrency, consistency, and lifetime

The Dapr actor model relies on two main components: the Dapr runtime, operating in the sidecar, and the Dapr placement service. We'll understand each of these in the following sections.

### Placement service

The placement service is responsible for keeping a map of the Dapr instances that are capable of serving actors. Considering our example, the `reservationactor-service` application is an example of such a service.

Once a new instance of our new `reservationactor-service` Dapr application starts, it informs the placement service that it is ready to serve actors of the `ReservationItemActor` type.

The placement service broadcasts a map – in the form of a hash table with the host's information and the served actor types – to all the Dapr sidecars operating in the environment.

Thanks to the host's map being constantly updated, actors are uniformly distributed over the actor service instances.

In the Kubernetes deployment mode of Dapr, the host is a **Pod** (a group of containers that are deployed together), while in standalone mode, the host is the local node itself. In Kubernetes, Pods can be terminated or initiated in response to many events (such as scaling out Pods and nodes, or a node being evicted from the cluster to be upgraded, added, or removed).

Considering the example that we deployed in a Kubernetes cluster, we should have at least three replicated Pods with `reservationactor-service` running together with the Dapr sidecar container. If our cookie-selling e-commerce site has about 300 active cookies' SKU, approximately 100 actors of the `ReservationItemActor` type will reside in each of the `reservationactor-service` Pods.

You can learn more about the placement service in the Dapr documentation at <https://docs.dapr.io/developing-applications/building-blocks/actors/actors-overview/#actor-placement-service>.

Now, let's learn how the Dapr actor model deals with concurrency and consistency.

## Concurrency and consistency

The actor model's implementation in Dapr approaches the state's concurrency manipulation by transparently enforcing turn-based access with **per-actor** locks. This lock is acquired at the beginning of each interaction and is released afterward.

The following diagram shows turn-based access for Dapr actors:

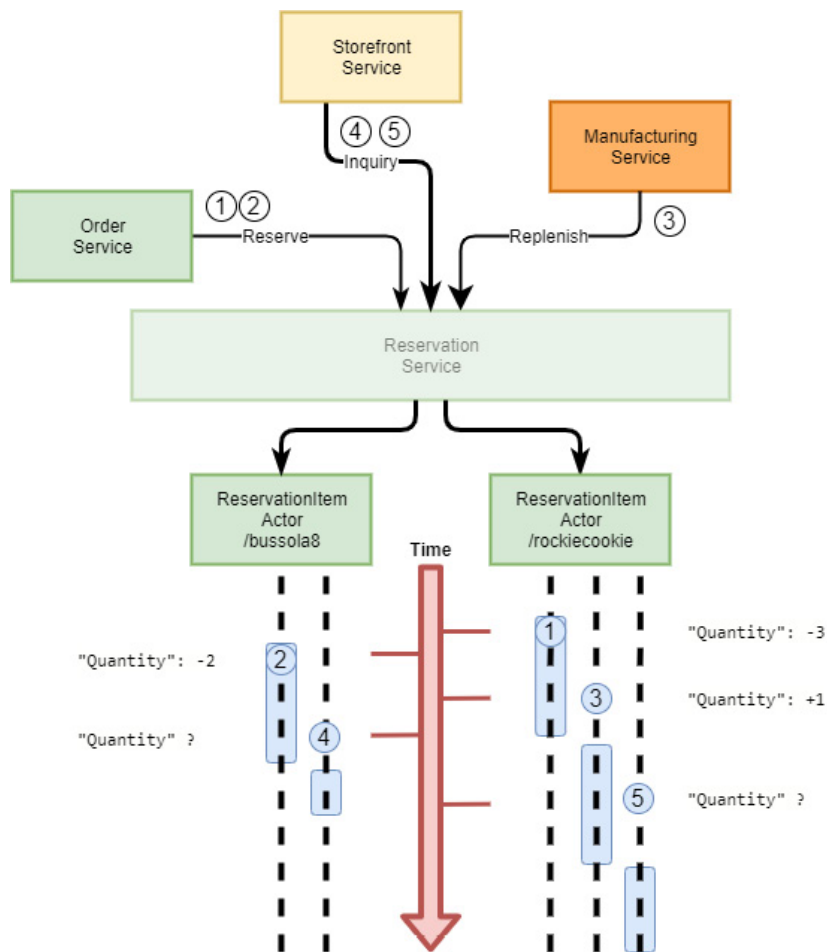


Figure 8.3 – Turn-based concurrency access to Dapr actors

In the preceding diagram, we can appreciate the impact of locks being acquired and released on a per-actor basis. Considering the two actors in this example – the `ReservationItemActor` actor for the **bussola8** SKU and the `ReservationItemActor` actor for the **rockiecookie** SKU – the following steps correspond to client (other Dapr or non-Dapr applications) requests in order of time, as follows:

1. A request to reserve the quantity on the actor with an ID of **rockiecookie**. The actor is ready, and no locks are present, so it is immediately granted, and the interaction can start immediately.
2. A request to reserve quantity on the actor with an ID of **bussola8**. This actor is ready with no locks present either, so the interaction can start immediately.
3. A request to replenish the quantity for actor ID **rockiecookie** is received. The actor is still locked since step 1, so the request waits for the lock to be released. Once the actor becomes available, the interaction can start.
4. A request to retrieve the current balance quantity for actor ID **bussola8** is received. The actor is still locked since step 2, so the request waits for it to be released. Once the actor becomes available, the quick (read) operation can happen.
5. A similar request reaches actor ID **rockiecookie**. Since all the interactions with the actor are treated equally and the actor is locked to working on step 3, the request waits for the lock's release to quickly interact with the actor.

None of these interactions required any additional effort from the Dapr application's code; everything is handled by the Dapr runtime.

While turn-based access to Dapr Actors imposes strict concurrency control, it's important to remember that this is enforced on a per-actor basis: concurrent requests to different actors can be fulfilled independently at the same time.

This approach gives Dapr an advantage in maintaining data consistency while acting as a cache. Since the actor's state could be as simple as a record in the Dapr state store, receiving data manipulation requests on a specific record with a serial approach puts any database in the best condition to operate.

From an individual actor's perspective, it's important to avoid executing long or variable operations as these would influence the lock's duration and impact client interaction. As an example, an actor should avoid I/O operations or any other blocking interaction.

The actor pattern is best used with many fast, independent actors.

From a client perspective, though, trying to coordinate or aggregate information from too many actors may not result in the best possible experience. As we learned in *Chapter 5, Introducing State Management*, it's possible to submit queries directly to the underlying database to extract or aggregate data from state stores.

In our specific sample, `reservation-service` evolves from being the Dapr application in charge of managing the `reservationitemstate` state to becoming the main client of the `ReservationItemActor` actor type, which is part of the `reservationactor-service` application.

Any Dapr-enabled application can interact with the Actors via a **software development kit (SDK)**, or by directly invoking the Dapr **application programming interface (API)** at the endpoint that was exposed by the Dapr sidecar.

At this stage in developing Actors in Dapr, an actor cannot directly subscribe to a publish/subscribe topic. This is an example in which an actor needs to be invoked indirectly from another service.

Let's briefly explore how the resiliency feature can improve the interactions between actor instances.

## Resiliency

The resiliency feature in Dapr allows us to configure how to handle transient errors, which are encountered by applications interacting with each other or with external components. As we learned in the *Resiliency* section in *Chapter 4, Service-to-Service Invocation*, a few types of policies can be applied to targets, including actors.

Resiliency is very useful once applied to the Actor building block; let's see why. With apps as the target, resiliency policies influence how to handle transient errors experienced by the invoking application so that the invoked application does not get bogged down by more unnecessary retries. Taking into consideration the constraints imposed by the locking mechanism in Actors, applying resiliency policies can be even more beneficial both to the actor instance and to the client Dapr applications.

A special trait of the resiliency feature that's applied to Actors is that the circuit breaker policy type can have a scope that involves all the actor instances of a certain type, each actor by its ID, or both.

There is still one major concept left to assimilate regarding the Dapr actor model: the actor's lifetime.

## Lifetime

Actors in Dapr are not explicitly created; they are brought to life by invoking them (say *Actor*, *Actor*, *Actor* three times in front of a mirror and they will appear!) instead.

In the scope of the Dapr .NET SDK, the actor's implementation can override the `OnActivateAsync` method from the base class to intercept the activation moment.

An actor will be deactivated after a period of inactivity. Deactivation is temporary as the Dapr runtime is ready to rehydrate it back into memory from the persisted state store if a new request arises. The idle timeout of each actor type can be configured. So, in your architecture, you can define both long-running actors and short-lived ones.

An actor instance can become aware of its deactivation if we override the `OnDeactivateAsync` method of the base class. Any interaction with the actor's instance extends its lifetime as it restarts the timeout clock.

Reminders and timers are two important features of Dapr actors. Both are useful for scheduling activity on an actor, whether it is recurring or intended to be executed only once. As an example, let's say you want to delay executing your service code from the initial request. The two differ in terms of their behavior, as highlighted here:

- Timers can trigger while the actor is active, but once deactivated, the timer is no longer effective, and it does not keep the actor active.
- Reminders are persistent, as they trigger the registered method even on a deactivated actor. Due to this, a reminder extends the actor's lifetime.

Now that we have learned about the lifetime, placement, consistency, and concurrency of the Dapr actor model, we are ready to apply this knowledge and implement the changes we've discussed so far in our small e-commerce solution.

## Implementing actors in an e-commerce reservation system

Equipped with information about the actor's pattern and with a plan to implement the evolution of our sample project by introducing Dapr actors, we now have several steps to complete, as follows:

1. Preparing the actor's projects
2. Implementing the actor's model
3. Accessing actors from other Dapr applications
4. Inspecting the actor's state

Let's start by creating the .NET projects.

### Preparing the Actor's projects

To implement actors with Dapr in C#, we must create an actor interface project that's separate from the actor's service implementation in two different projects.

The actor's interface will be referenced by the other services or clients that need to interact with the actors. Let's create the interface project, as follows:

```
PS C:\Repos\dapr-samples\chapter07> dotnet new classlib -o
sample.microservice.reservationitemactor.interfaces
PS C:\Repos\dapr-samples\chapter07> cd .\sample.microservice.
reservationactor.interfaces\
PS C:\Repos\dapr-samples\chapter08\sample.microservice.
reservationactor.service> dotnet add package Dapr.Actors -v
1.7.0
```

The following command is going to create the project for the reservationactor-service Dapr application:

```
PS C:\Repos\dapr-samples\chapter07> dotnet new webapi -o
sample.microservice.reservationitemactor.service
PS C:\Repos\dapr-samples\chapter07> cd .\sample.microservice.
reservationactor.service\
PS C:\Repos\dapr-samples\chapter08\sample.microservice.
reservationactor.service> dotnet add reference ..\sample.
microservice.reservationactor.interfaces\sample.microservice.
reservationactor.interfaces.csproj
PS C:\Repos\dapr-samples\chapter08\sample.microservice.
reservationactor.service> dotnet add package Dapr.Actors -v
1.7.0
PS C:\Repos\dapr-samples\chapter08\sample.microservice.
reservationactor.service> dotnet add package Dapr.Actors.
AspNetCore -v 1.7.0
```

Each project also refers to the .NET SDK for Actors, located in the `Dapr.Actors` package. The `reservationactor-service` project refers to the `Dapr.Actors.AspNetCore` package for Actors in ASP.NET.

Now that our projects are ready, we can implement the actors.

## Implementing the actor's model

Let's start by implementing the actor interface in the `IReservationItemActor.cs` class of the `sample.microservice.reservationitemactor.interfaces` project, as follows:

```
using Dapr.Actors;
namespace sample.microservice.reservationactor.interfaces;
public interface IReservationItemActor : IActor
{
    Task<int> AddReservation(int quantity);
    Task<int> GetBalance();
    Task RegisterReminder();
    Task UnregisterReminder();
    Task RegisterTimer();
    Task UnregisterTimer();
}
```

The `IReservationItemActor` interface, which inherits from `IActor` in `Dapr.Actors`, is as simple as the tasks our actor will fulfill: `AddReservation` will receive a reservation for an SKU, while `GetBalance` will return its available quantity.

The other methods in `IReservationItemActor` are present in the code to showcase the `Reminder` and `Timer` features of Dapr Actors, which we briefly described in the previous section, *Actor concurrency, consistency, and lifetime*. To learn more about `Reminder` and `Timer`, I recommend the Dapr documentation at <https://docs.dapr.io/developing-applications/building-blocks/actors/howto-actors/>.

Let's move on to the implementation in the `ReservationItemActor.cs` class in the `sample.microservice.reservationactor.service.service` project, as follows:

```
namespace sample.microservice.reservationactor.service;
internal class ReservationItemActor : Actor,
    IReservationItemActor, IRemindable
{
    public const string StateKey = "reservationitem";
    public ReservationItemActor(ActorHost host)
        : base(host)
    {
    }
    protected override Task OnActivateAsync()
    {
        Console.WriteLine($"Activating actor id: {this.Id}");
        return Task.CompletedTask;
    }
    protected override Task OnDeactivateAsync()
    {
        Console.WriteLine($"Deactivating actor id: {this.Id}");
        return Task.CompletedTask;
    }
    ... class continue below ...
}
```

In the previous section of the `ReservationItemActor.cs` file, we can see a typical definition of a Dapr actor type in C#: the `ReservationItemActor` class derives from `Dapr.Actors.Runtime.Actor`, which implements the interface of our `IReservationItemActor` actor interface and the `IRemindable` interface from the `Dapr.Actors.Runtime` namespace. The latter interface is used to enable **reminders**, a powerful feature that's used to influence the actor life cycle.

The `ReservationItemActor.cs` file continues like this:

```
... class continue from above ...
    public async Task<int> AddReservation(int quantity)
    {
        var SKU = this.Id.GetId();
        var state = await this.StateManager.
            TryGetStateAsync<ItemState>(StateKey);
    ... omitted ...
        await this.StateManager.SetStateAsync<ItemState>(
            StateKey, value);
        Console.WriteLine($"Balance of {SKU} was
            {initialBalanceQuantity}, now
            {value.BalanceQuantity}");
        return value.BalanceQuantity;
    }
    public async Task<int> GetBalance()
    {
        var state = await this.StateManager.
            GetStateAsync<ItemState>(StateKey);
        return state.BalanceQuantity;
    }
    ... omitted ...
}
```

In the method's implementation, it's worth noting how an instance can access the state store via the Dapr actor object model.

With the `TryGetStateAsync<type>(StateKey)` method of `StateManager`, which is implemented in the `Dapr.actors.runtime.Actor` base class, the actor attempts to retrieve a state with a `StateKey` key from the store. The state might not exist yet if the actor instance has only just been implicitly created. Alternatively, you can use `StateManager.GetStateAsync<type>(StateKey)`.

With the `StateManager.SetStateAsync<type>(StateKey, value)` method, you inform the Dapr runtime to save the serializable value in the state element with the `StateKey` key, after the actor method completes successfully.

Our actor implementation is not ready yet; the actor type we just implemented must be registered with the Dapr runtime. The following is from the `Program.cs` file, which is leveraging the ASP.NET 6 Minimal Hosting feature:

```
var builder = WebApplication.CreateBuilder(args);
var jsonOpt = new JsonSerializerOptions()
{
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
    PropertyNameCaseInsensitive = true,
};
builder.Services.AddActors(options =>
{
    options.Actors.RegisterActor<ReservationItemActor>();
    options.JsonSerializerOptions = jsonOpt;
});
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
var app = builder.Build();
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
app.MapActorsHandlers();
app.Run();
```

As you can see, an actor service has a different implementation than a standard service in Dapr, also from a configuration perspective. `builder.Services.AddActors` adds support for the Actor runtime, while `options.Actors.RegisterActor` registers the actor of the `ReservationItemActor` type. In the end, the `MapActorsHandlers` method registers the Dapr-specific routes for Actors.

The ASP.NET project that's implementing our Dapr Actors is now ready to be made reachable by the other Dapr applications.

## Accessing actors from other Dapr applications

First, we will create a separate project that will contain the actor interface so that we can reference it from clients.

As depicted in *Figure 8.2*, we decided to keep the interaction with actors of the `ReservationItemActor` type in the scope of `reservationactor-service`. We will add the reference here, as follows:

```
PS C:\Repos\dapr-samples\chapter08> cd .\sample.microservice.
reservation\
PS C:\Repos\dapr-samples\chapter08\sample.microservice.
reservation> dotnet add reference ..\sample.microservice.
reservationactor.interfaces\sample.microservice.
reservationactor.interfaces.csproj
```

We should also add the reference to the `Dapr.Actors` package in `reservationactor-service`, as we did in the previous projects.

In the previous chapters, our code directly accessed the state store information via the `DaprClient` instance, which is what our ASP.NET controllers have been injected with by Dapr. The change we will apply to `ReservationController.cs` will impact this part:

```
... omitted ...
[Topic(PubSub, common.Topics.OrderSubmittedTopicName)]
[HttpPost(common.Topics.OrderSubmittedTopicName)]
public async Task<ActionResult<OrderReservation>>
ReserveOrder(Order order, [FromServices] DaprClient daprClient)
{
    var stateReservation = await daprClient.
        GetStateEntryAsync<ReservationState>(StoreName_reservation,
            order.Id.ToString());
    stateReservation.Value ??= new ReservationState() { OrderId =
        order.Id, ReservedItems = new List<ItemReservation>() };
    var result = new OrderReservation() { OrderId = order.Id,
        ReservedItems = new List<Item>() };
    foreach (var item in order.Items)
    {
        var SKU = item.ProductCode;
        var quantity = item.Quantity;
        var actorID = new ActorId(SKU);
        var proxy = ActorProxy.Create<IReservationItemActor>
            (actorID, "ReservationItemActor");
        var balanceQuantity = await proxy.
            AddReservation(quantity);
    }
}
```

```

        result.ReservedItems.Add(new Item{SKU = SKU,
            BalanceQuantity = balanceQuantity});
    }
    ... omitted ...

```

There are a few things to examine in the previous code snippet. By instantiating the `ActorId (SKU)` type, we pass it as a parameter in `ActorProxy.Create<IReservationItemActor>(actorID, "ReservationItemActor")`. We are instructing the Dapr runtime to look for – in the map kept in sync by the Dapr placement service – the actor service instance that's responsible for handling the specific actor with the key equal to the SKU.

The instance of `Dapr.actors.Client.ActorProxy` we just created lets us invoke the actor by leveraging its interface as if it were a local object. The `proxy.AddReservation(quantity)` method is what we defined previously in the `IReservationItemActor` interface and implemented in `ReservationItemActor`.

All the concepts we described previously in this chapter hide behind this simple object model – isn't it nice, elegant, and simple?

We can launch the Dapr applications as we did previously, with the **command-line interface (CLI)** or **Tye**, now with the addition of the newly created `reservationactor-service` Actor service from the `sample.microservice.reservationactor.service.csproj` project, as follows:

```

dapr run --app-id "reservationactor-service" --app-port "5004"
--dapr-grpc-port "50040" --dapr-http-port "5014" --components-
path "./components" -- dotnet run --project ./sample.
microservice.reservationactor.service/sample.microservice.
reservationactor.service.csproj --urls="http://+:5004"

```

To appreciate the changes that have been applied to the architecture, we can submit an order, as described in the `order.test.http` test file, by invoking the ASP.NET controller for `order-service`, which is currently configured to be exposed at `http://localhost:5001/order`, or by reaching it via the Dapr sidecar at `http://localhost:5010/v1.0/invoke/order-service/method/order`.

The following is the output from `reservation-service`:

```

== APP == Reservation in d6082d80-1239-45db-9d35-95e587d7b299
of rockiecookie for 4, balance 32
== APP == Reservation in d6082d80-1239-45db-9d35-95e587d7b299
of bussola8 for 7, balance 59
== APP == Reservation in d6082d80-1239-45db-9d35-95e587d7b299
of crazycookie for 2, balance 245

```

```
== APP == Reservation in d6082d80-1239-45db-9d35-95e587d7b299
completed
```

Let's focus on the output from `reservationactor-service`, shown here:

```
== APP == Actor: rockiecookie Activated
== APP == Balance of rockiecookie was 36, now 32
== APP == Activating actor id: bussola8
== APP == Actor: bussola8 Activated
== APP == Balance of bussola1 was 6, now 4
== APP == Balance of crazycookie was 247, now 245
```

Here, we can see that some actors have been implicitly activated by us interacting with them, while other ones were already active in the host's memory.

In the previous test interaction, `reservation-service`, reached by `order-service`, invoked the Dapr application's `reservationactor-service` by leveraging the .NET SDK. Even though we do not manually compose URLs, it is useful to understand the different syntax to interact with the Actor building block.

In *Chapter 4, Service-to-Service Invocation*, we understood that the Dapr service-to-service building block follows this pattern in composing URLs:

```
http://localhost:3500/v1.0/invoke/app-id/method/methodname
```

The `app-id` and `methodname` elements are the name of the Dapr application we need to reach and the name of the method we want to invoke, respectively.

Similarly, in *Chapter 5, Introducing State Management*, we understood how the Dapr state management building block adopts the following pattern:

```
http://localhost:3500/v1.0/state/storename/key
```

The `storename` element is the name of a configured state store, and `key` is used to identify the key/value pair of interest.

Now, let's observe the more complex URL pattern adopted by the Actor building block:

```
http://localhost:3500/v1.0/actors/actortype/actorid/method/
methodname
```

The previous URL invokes a method of an instance. The `actortype` element specifies the type of the Actor, `actorid` identifies the specific instance, and `methodname` is the name of the method we want to invoke on the Actor instance.

To interact with the state, the Actor building block relies on the following URL pattern:

```
http://localhost:3500/v1.0/actors/actortype/actorid/state/key
```

The key element identifies the key/value pair of interest in the state store. In the Actor building block, only one state store can be configured, and no element is needed in the URL.

Now that we have proved how easy it is to introduce the actor model to an existing Dapr application, we should verify how our actors' data gets persisted.

## Inspecting the actor's state

So far, we've learned how to configure a Dapr state store to make it suitable for actors. It is interesting to note how the state key is composed, as shown by the following item in Azure Cosmos DB, which corresponds to an actor's state:

```
{
  "id": "reservationactor-service||ReservationItemActor||
    cookie2||reservationitem",
  "value": {
    "BalanceQuantity": -12,
    "Changes": [
      {
        "Quantity": 12,
        "ReservedOn": "2022-04-
          09T22:17:47.0511873Z",
        "SKU": "cookie2"
      }
    ],
    "SKU": "cookie2"
  },
  "partitionKey": "reservationactor-service||ReservationItem
    Actor||cookie2",
  "_rid": "h+ETAIFd00cBAAAAAAAAA==",
  "_self": "dbs/h+ETAA==/colls/h+ETAIFd00c=/docs/
    h+ETAIFd00cBAAAAAAAAA==/",
  "_etag": "\"0000dd23-0000-0d00-0000-5f80e18a0000\"",
  "_attachments": "attachments/",
  "_ts": 1602281866
}
```

---

As you can see, the key has been composed with the `<application ID>||<actor type>||<actor Id>||<key>` pattern, where the following apply:

- `reservationactor-service` is the application ID.
- `actor Id` is the actor's **unique identifier (UID)** that we chose to adopt as the SKU. Here, this is `cookie2`.
- `ReservationItemActor` is the actor type.
- `key` identifies the state key that's used, which is `reservationitem` here. An actor can have multiple states.

With that, we have completed our journey of learning about the actor model in Dapr.

## Summary

In this chapter, we learned that the actor model that's supported by Dapr is a very powerful tool in our toolbox.

We understood the scenarios that benefit the most from applying actors, and how to avoid the most common implementation pitfalls.

By configuring Dapr Actors, from the state store to the ASP.NET perspective, we appreciated how the simplicity of Dapr extends to this building block too.

Next, we introduced an actor type to our existing architecture. By doing so, we learned how to separate the contract (interface) from the implementation and invoke it from other Dapr services.

Note that this is another example of how Dapr facilitates the development of microservice architectures by addressing the communication and discovery of services (how easy is it for a client to interact with an actor?). It also unleashes the independent evolution of our architecture's components; introducing actors in our example has been seamless for the rest of the services. With this new information and experience, you will be able to identify how to best introduce Dapr Actors into your new or existing solution.

Now that our sample solution is complete, we are ready to move it to an environment that's capable of exposing it to the hungry and festive customers of *Biscotti Brutti Ma Buoni*. Starting from the next chapter, we will learn how to prepare the Dapr-based solution so that we can deploy it on Kubernetes.



# Part 3: Deploying and Scaling Dapr Solutions

Now that we know how Dapr works and have learned how to create applications with Dapr, it is time to deploy to Kubernetes and load-test the applications. We will also learn how to leverage Dapr with the richness of Kubernetes but without its complexity.

This part contains the following chapters:

- *Chapter 9, Deployment to Kubernetes*
- *Chapter 10, Exposing Dapr Applications*
- *Chapter 11, Tracing Dapr Applications*
- *Chapter 12, Load-Testing and Scaling Dapr*
- *Chapter 13, Leveraging Serverless Containers with Dapr*



# Deploying to Kubernetes

In this chapter, we will shift our focus to the Kubernetes hosting mode known as Dapr. First, we will learn how to prepare our sample projects so that they can be deployed in a containerized form, before moving on to preparing Dapr in the Azure Kubernetes Service cluster.

This chapter will help us gain visibility of the production-ready environment for a solution based on Dapr: a Kubernetes cluster.

In this chapter, we will cover the following topics:

- Setting up Kubernetes
- Setting up Dapr on Kubernetes
- Deploying a Dapr application to Kubernetes
- Exposing Dapr applications to external clients

As architects and developers, we usually mainly focus on defining and implementing a solution. However, it is important to understand the implications of the deployment options and how these can influence our design choices, as well as impact our overall solution architecture.

Our first objective is to provision the Kubernetes cluster and connect to it from our development environment.

## Technical requirements

The code for this chapter can be found on GitHub at <https://github.com/PacktPublishing/Practical-Microservices-with-Dapr-and-.NET-Second-Edition/tree/main/chapter09>.

In this chapter, the working area for the scripts and code can be found at `<repository path>\chapter09\`. In my local environment, it can be found here: `C:\Repos\dapr-samples\chapter09`.

Please refer to the *Setting up Dapr* section of *Chapter 1, Introducing Dapr*, for a complete guide on the tools needed to develop with Dapr and work with the examples provided.

## Setting up Kubernetes

While the discussion around microservice architectures has evolved independently, the concept of containerized deployments has propelled its popularity among developers and architects.

Once you start to have a multitude of microservices, each comprised of one or many containers, you soon realize you need a piece of software that can orchestrate these containers. In a nutshell, orchestration is the reason why Kubernetes is so relevant and frequently appears in the context of microservice architectures.

### Important note

**Kubernetes (k8s)** is the most popular open source container orchestrator and is a project that's maintained by the **Cloud Native Computing Foundation (CNCF)**. To learn more about Kubernetes, I suggest that you read straight from the source at <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>.

In this section, we are going to provision an **Azure Kubernetes Service (AKS)** cluster. Even if it is not in this book's scope to learn all the details of AKS, for those of you who are not already Kubernetes geeks, it will be helpful for you to become familiar with some of its concepts and tooling.

These are the steps we will be going through:

1. Creating an Azure resource group
2. Creating an AKS cluster
3. Connecting to the AKS cluster

For your convenience, all the commands in this step and the following ones have been provided in the `Deploy\setup-AKS.ps1` file.

Let's start by preparing a resource group.

### Creating an Azure resource group

In a Windows terminal, log in to Azure by using the Azure CLI. We could also use the portal here but given that you will be using files from the repository we cloned locally in the next few sections, it might be easier for you to run the CLI locally. This helps us keep a consistent experience between Azure and Kubernetes.

Let's connect to the subscription that we want to provision the cluster in. It will probably be the same Azure subscription you used in the previous chapters. Mine is named `Sandbox`:

```
PS C:\Repos\dapr-samples\chapter09> az login
PS C:\Repos\dapr-samples\chapter09> az account set -
subscription "Sandbox"
```

All the commands that we will be issuing via the Azure CLI will be issued in the context of the specified Azure subscription.

```
PS C:\Repos\dapr-samples\chapter09> az group create -name
$aksname --location $location
```

In the previous command, we must create the resource group with the `$name` and `$location` parameters of our choice.

## Creating an AKS cluster

Now, we can create the AKS cluster. Please check the walkthrough available at <https://docs.microsoft.com/en-us/azure/aks/kubernetes-walkthrough> for additional information.

In the following CLI command, we are choosing to enable the monitoring feature and prefer the use of `VirtualMachineScaleSets` (an Azure feature that lets you manage and scale nodes as a group) for the nodes rather than a simple VM:

```
PS C:\Repos\dapr-samples\chapter09> az aks create --resource-
group $rg --name $aksname `
    --node-count 3 --node-vm-size Standard_D2s_v3 `
    --enable-addons monitoring `
    --vm-set-type VirtualMachineScaleSets `
    --generate-ssh-keys
```

After waiting a few minutes for the cluster to be created, we can verify the status of the AKS cluster resource, named per our choice in variable `$aksname`, with the following command:

```
PS C:\Repos\dapr-samples\chapter09> az aks show --name
$aksname --resource-group $rg
```

Once we have successfully created the AKS cluster, we can connect to it.

## Connecting to the AKS cluster

Once the cluster has been created, we need to install the Kubernetes tools on our development environment – namely, the **kubectl** CLI, which is facilitated by the Azure CLI – with the following command:

```
PS C:\Repos\dapr-samples\chapter09> az aks install-cli
```

With the Azure CLI, we can also retrieve the credentials we need to gain administrative access to the cluster. These credentials will be merged into the default location for the Kubernetes configuration file:

```
PS C:\Repos\dapr-samples\chapter09> az aks get-credentials
--name $aksname --resource-group $rg
Merged "xyz" as current context in C:\Users\user\.kube\config
```

We now have access to the cluster, and with the `kubectl` CLI, we can control any aspect of it. As a starting point, let's examine the cluster's composition:

```
PS C:\Repos\dapr-samples\chapter09> kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
aks-nodep				
o011-14504866-vmss000003	Ready	agent	11m	v1.21.9
aks-nodep				
o011-14504866-vmss000004	Ready	agent	11m	v1.21.9
aks-nodep				
o011-14504866-vmss000005	Ready	agent	11m	v1.21.9

According to the `kubectl get nodes` command, we have three nodes running, as specified in the AKS provisioning command.

From this point forward, the experience will have less to do with Azure and more to do with Dapr and Kubernetes. Due to this, it will apply to any suitable similar containerized environment, such as other cloud providers or edge/hybrid scenarios. I suggest getting familiar with the `kubectl` CLI documentation, which is available at <https://kubernetes.io/docs/reference/kubectl/>.

Additionally, you can install the client tools for **Helm** in your development environment. Helm is a package manager for Kubernetes, often used for more rich and complex solutions. To learn more, look at the documentation at <https://helm.sh/docs/intro/install/>.

In this section, we managed to create an AKS instance, install the necessary Kubernetes tools locally, and gain access to the cluster itself. The next step is to install Dapr in our AKS cluster.

## Setting up Dapr on Kubernetes

At this stage, a Kubernetes cluster (specifically, AKS on Azure) is ready to accommodate our workload. We need to install Dapr before we can move on to the preparation phase for our applications.

In *Chapter 1, Introducing Dapr*, we used the `dapr init` CLI command to initialize Dapr in our development environment. We'll use it again here to initialize Dapr in Kubernetes, but we will add the `-k` parameter:

```
PS C:\Repos\dapr-samples\chapter09> dapr init -k
Making the jump to hyperspace...
Note: To install Dapr using Helm, see here: https://docs.dapr.io/getting-started/install-dapr/#install-with-helm-advanced
Deploying the Dapr control plane to your cluster...
Success! Dapr has been installed to namespace dapr-system.
To verify, run `dapr status -k` in your terminal.
To get started, go here: https://aka.ms/dapr-getting-started
```

The preceding command installs and initializes the Dapr components in the cluster that corresponds to the current Kubernetes configuration.

We can verify that the Dapr service we learned about in *Chapter 1, Introducing Dapr*, is now present in the cluster by executing the following command in the Dapr CLI:

```
PS C:\Repos\dapr-samples\chapter09> dapr status -k
```

NAME	NAMESPACE	HEALTHY
STATUS REPLICAS VERSION AGE CREATED		
dapr-dashboard	dapr-system	True Running
1 0.10.0 16d ...		
dapr-placement-server	dapr-system	True Running
1 1.8.4 16d ...		
dapr-sentry	dapr-system	True Running
1 1.8.4 16d ...		
dapr-sidecar-injector	dapr-system	True Running
1 1.8.4 16d ...		
dapr-operator	dapr-system	True Running
1 1.8.4 16d ...		

The `dapr status -k` command is equal to querying the currently running Pods in the Kubernetes `dapr-system` namespace via the `kubectl` CLI:

```
PS C:\Repos\dapr-samples\chapter09> kubectl get pods -n dapr-system
```

NAME	READY	STATUS
RESTARTS AGE		
dapr-dashboard-78557d579c-ng vqt 1/1	Running	
0 2m4s		
dapr-operator-74cdb-5fff9-7qt89 1/1	Running	0 2m4s
dapr-placement-7b5bbdd95c-d6kmw 1/1	Running	0 2m4s
dapr-sentry-65d64b7cd8-v7z9n 1/1	Running	0 2m4s
dapr-sidecar-injector-7759b8b9c4-whvph 1/1	Running	0 2m4s

From the **Pods** count in the preceding output, you can see that there is only one replica of each Pod from the Dapr system services.

However, this could change with the highly available deployment option of Dapr. For our development environment, it's fine to have just one replica since our cluster has a reduced number of nodes. For this and other production guidelines on how to set up and operate Dapr on Kubernetes in a production environment, please see the documentation at <https://docs.dapr.io/operations/hosting/kubernetes/kubernetes-production/>.

Let's explore the Kubernetes services provided by Dapr by running the following command in the context of the `dapr-system` namespace:

```
PS C:\Repos\dapr-samples\chapter09> kubectl get services -n dapr-system
```

NAME	TYPE	CLUSTER-IP	EXTER
NAL-IP	PORT(S)	AGE	
dapr-api	Clus		
terIP	10.0.123.133	<none>	80/TCP 23d
dapr-dashboard	Clus		
terIP	10.0.206.23	<none>	8080/TCP 23d
dapr-placement-server	Clus		
terIP	None	<none>	50005/TCP,8201/TCP 23d
dapr-sentry	Clus		
terIP	10.0.143.67	<none>	80/TCP 23d
dapr-sidecar-injector	Clus		
terIP	10.0.36.244	<none>	443/TCP 23d
dapr-webhook	Clus		
terIP	10.0.166.229	<none>	443/TCP 23d

You can also gather the same information from the Azure portal, as shown in the following screenshot:

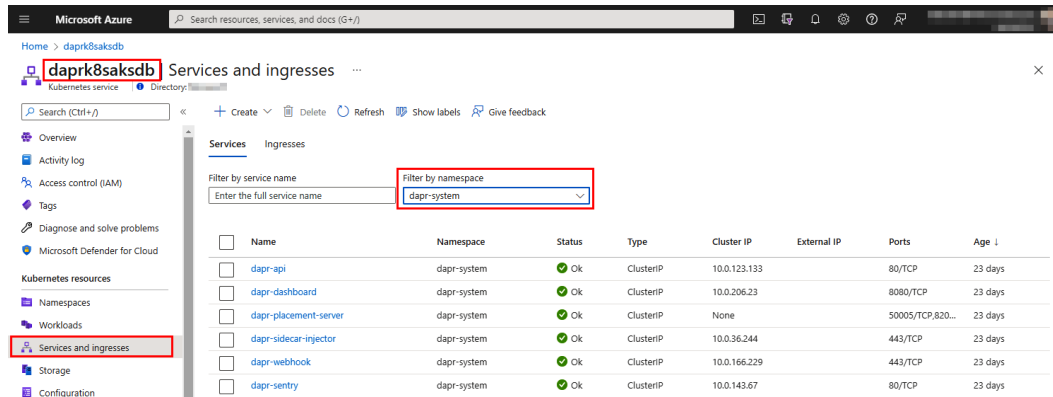


Figure 9.1 – Services by namespace in AKS

Here, upon inspecting the Kubernetes resources of the AKS cluster, you will see **Services and ingresses** that have been configured in the namespace.

We can also leverage the Dapr dashboard here:

```
PS C:\Repos\dapr-samples\chapter09> dapr dashboard -k
Dapr dashboard found in namespace:      dapr-system
Dapr dashboard available at:      http://localhost:8080
```

With the `dapr dashboard -k` parameter, we can open the Dapr dashboard that is running in the Kubernetes environment. All this is done inside the `dapr-system` namespace:

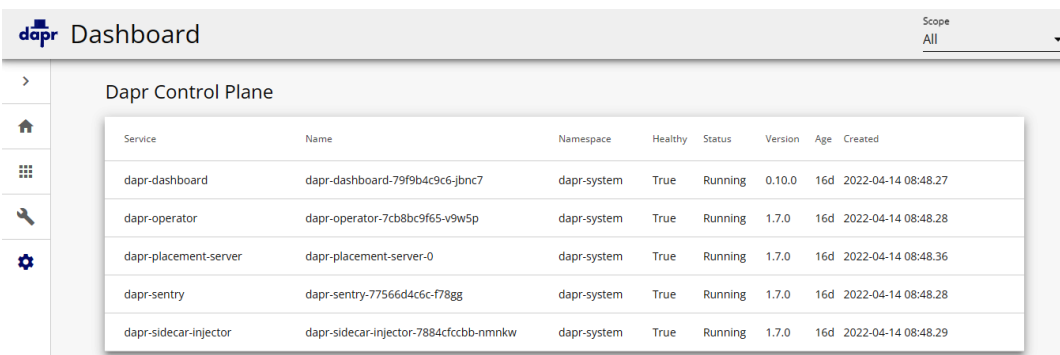


Figure 9.2 – The Dapr dashboard in Kubernetes hosting mode

As a recap, you can find an overview of the various methods (kubect!, the Azure portal, the Dapr dashboard, and so on) we can use to gather feedback on the Dapr services running in Kubernetes by going to the following Kubernetes section of the Dapr documentation: <https://docs.dapr.io/operations/hosting/kubernetes/>.

The Kubernetes cluster is now ready to accept the deployment of a Dapr-enabled application.

## Deploying a Dapr application to Kubernetes

The service code for our Dapr application is now complete. However, we must package it so that it can be deployed to Kubernetes appropriately. Our first objective is to publish these services as Docker containers.

The sample that's available for this chapter, `C:\Repos\dapr-samples\chapter09`, is aligned with the status we reached at the end of *Chapter 8, Using Actors*. To recap, the following Dapr applications comprise our overall solution:

- `sample.microservice.order`
- `sample.microservice.reservation.service`
- `sample.microservice.reservationactor.service`
- `sample.microservice.customization`
- `sample.microservice.shipping`

The preceding list represents the Dapr applications, each with its corresponding ASP.NET project, that we need to build as Docker images to deploy to Kubernetes later.

While this chapter also shows how to build our Dapr applications into Docker images and push those images into a private container registry, ready-to-use images are available on **Docker Hub** if you wish to expedite them:

- `davidebedin/sample.microservice.order`, which corresponds to the `sample.microservice.order` application
- `davidebedin/sample.microservice.reservation`, which corresponds to the `sample.microservice.reservation.service` application
- `davidebedin/sample.microservice.reservationactor`, which corresponds to the `sample.microservice.reservationactor.service` application
- `davidebedin/sample.microservice.customization`, which corresponds to the `sample.microservice.customization` application
- `davidebedin/sample.microservice.shipping`, which corresponds to the `sample.microservice.shipping` application

Even if you decide to use the readily available container images, I suggest reading the instructions in the next two sections to get an overview of the process to build and push container images. Deployment scripts for leveraging the images on Docker Hub are available in this chapter's working area.

**What is Docker Hub?**

Docker Hub is the world's largest repository of container images, a service provided by Docker for finding and sharing container images with everyone. You can learn more at <https://hub.docker.com/>.

Next, we will start building the Docker images.

## Building Docker images

As we intend to deploy our sample application to Kubernetes, the Docker container is the deployment format we must (and will) use.

**Important note**

For more information on how to publish ASP.NET projects with the Docker container format, I suggest that you read the documentation at <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/docker/building-net-docker-images?view=aspnetcore-6.0>.

A Dockerfile is a text file that contains all the commands the Docker CLI needs to step through to build a Docker image. Let's start by examining the Dockerfile in the `sample.microservice.reservationactor.service` application folder:

```
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["sample.microservice.reservationactor.interfaces/
sample.microservice.reservationactor.interfaces.csproj",
      "sample.microservice.reservationactor.interfaces/"]
COPY ["sample.microservice.reservationactor.service/
sample.microservice.reservationactor.service.csproj",
      "sample.microservice.reservationactor.service/"]
RUN dotnet restore "sample.microservice.reservationactor.
interfaces/sample.microservice.reservationactor.interfaces.
csproj"
RUN dotnet restore "sample.microservice.reservationactor.
service/sample.microservice.reservationactor.service.csproj"
```

```

COPY . .
WORKDIR "/src/sample.microservice.reservationactor.service"
RUN dotnet publish "sample.microservice.reservationactor.
service.csproj" -c Release -o /app/publish --no-restore
FROM mcr.microsoft.com/dotnet/aspnet:6.0
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "sample.microservice.reservationactor.
service.dll"]
... omitted ...

```

In the previous Dockerfile, there are several stages in the build process. Here, we are using two separate base images:

- `mcr.microsoft.com/dotnet/aspnet:6.0` is an image containing the runtime and libraries for ASP.NET. It is optimized for running ASP.NET applications in production.
- `mcr.microsoft.com/dotnet/sdk:6.0` contains the .NET CLI, in addition to its runtime, and is suitable for building ASP.NET projects.

First, a stage based on the `mcr.microsoft.com/dotnet/sdk:6.0` image with the `build` alias is used as a destination for copying the involved projects, restoring the dependencies with `dotnet restore`, and then publishing them with `dotnet publish`.

In another, final, stage based on the `mcr.microsoft.com/dotnet/aspnet:6.0` base image, the content of the output of `publish` is copied from the `build` image into the root folder; `ENTRYPOINT` is declared to run the `dotnet` command on the project library once the container has been started.

The `Deploy\build-container.ps1` file has been prepared to build all the container images for our microservices. At this stage, let's examine the parameters of the `docker build` command:

```

... omitted ...
$container = "sample.microservice.order"
$latest = "{0}/{1}:{2}" -f $registry, $container, $default
$versioned = "{0}/{1}:{2}" -f $registry, $container,
$buildversion
docker build . -f .\sample.microservice.order\Dockerfile -t
$latest -t $versioned -build-arg BUILD_DATE=$builddate -build-
arg BUILD_VERSION=$buildversion
... omitted ...

```

In the previous snippet, the `docker build` command with the `-f` parameter references the Dockerfile of the Dapr application in the project folder, with `-t` setting two tags for the image while following the `registry/container:tag` pattern. Here, `registry` could be the name of a container registry, such as **Azure Container Registry (ACR)**, or the username in Docker Hub. Finally, the `--build-arg` parameters specify some arguments to be used in the Dockerfile itself.

Let's execute this `docker build` command directly in the command line:

```
PS C:\Repos\practical-dapr\chapter09> docker build . -f .\
sample.microservice.reservationactor.service\Dockerfile -t
sample.microservice.reservationactor.service:2.0
...
=> [build 1/9] FROM mcr.microsoft.com/dotnet/sdk:6.0@sha256:
fde93347d1cc74a03f1804f113ce85add00c6f0af15881181165ef04b
c76bd00                                0.0s
  => [stage-1 1/3] FROM mcr.microsoft.com/dotnet/aspnet:6.0@
sha256:431d21f51d76da537d305827e791d23bfcf4aebfa019c12ee8e14df
b71c64cca                                0.0s
...
  => [build 7/9] COPY . .
0.2s
  => [build 8/9] WORKDIR /src/sample.microservice.reservation
actor.service                            0.1s
  => [build 9/9] RUN dotnet publish "sample.microservice.
reservationactor.service.csproj" -c Release -o /app/publish
--no-restore                             6.5s
  => [stage-1 3/3] COPY --from=build /app/publish            0.1s
...
  => => writing image sha256:98ada5ed54d4256392a
fac9534745b4efe7d483ff597501bfef30c6645edb557            0.0s
  => => naming to docker.io/sample.microservice.reservation
actor.service:2.0    /sample.microservice.reservationactor.
service:2.0
```

In the previous output, most of which has been omitted for brevity, it's worth noting that each of our Dockerfile instructions is evaluated as a numbered step. In the end, we have our `ENTRYPOINT`.

The build process with Docker must be performed for each of the Dapr applications we intend to deploy to Kubernetes.

We should now have our images built and available in our development machine. At this point, we need to publish them to a container image registry so that we can use them from our Kubernetes cluster.

## Pushing Docker images

In this section, we are going to run our Dapr applications on the Kubernetes cluster. For this, we need to push the container images from our local environment to a location that can be accessed by the Kubernetes cluster.

A Kubernetes cluster can retrieve the images from a **container registry**, which usually offers a public and/or private container repository. A **container repository** is a collection of different versions of a container:

- **Docker Hub** is a container registry for private or public repositories
- **Azure Container Registry (ACR)** is a private repository for containers running on Azure
- Other container registry options may be available in private and public spaces

I decided to use ACR because it fits well with the overall Azure-focused scenario. If you want, you can publish the sample containers to public or private repositories on Docker Hub, or any other registry.

For your convenience, the commands in this section can be found in the `Deploy\setup-ACR-AKS.ps1` file; please remember to adapt the parameters to your specific case.

For a walkthrough on how to create an ACR instance, please take a look at the documentation provided at <https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-portal>.

### Important note

VS Code offers a rich developer experience when it comes to building Docker containers and pushing them to remote registries. Here, it makes sense to leverage these features while learning about something new, but it is highly recommended to integrate the container build's process into a CI/CD pipeline while leveraging GitHub, Azure DevOps, or any other platform suitable for the task.

As a starting point, I suggest that you read this article by my colleague Jessica Tibaldi on a CI/CD approach to Dapr: <https://www.linkedin.com/pulse/s01e02-dapr-compass-how-we-organized-work-jessica-tibaldi/>.

If we examine our local environment, we should note that we have some new Docker container images. The following is a screenshot from VS Code:

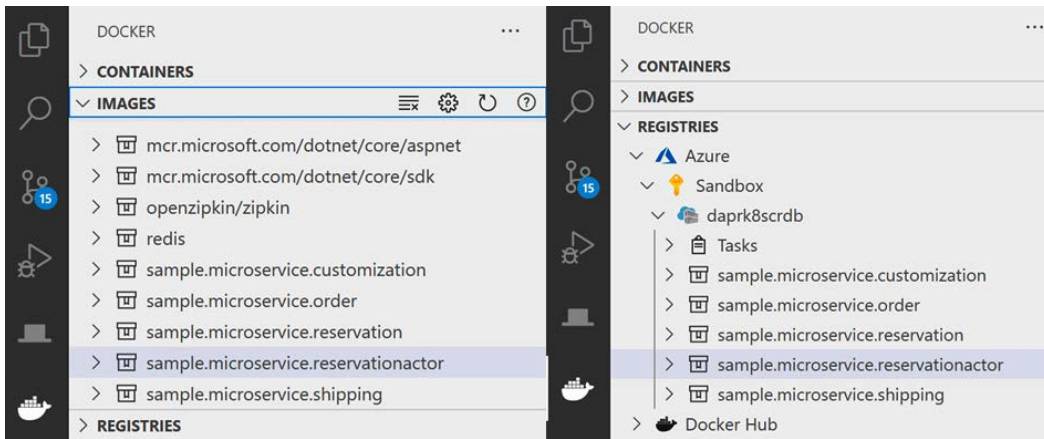


Figure 9.3 – Local images and ACR in VS Code

On the left-hand side of the preceding screenshot, we can see the images that are known by our Docker local environment. All the container images we just built are present in this list, starting with the `sample.microservice.reservationactor.service` image that we analyzed in the previous section.

On the right-hand side of the preceding screenshot, we can see the destination of the container images: an ACR instance.

The Azure extension in VS Code offers an integrated developer experience by easing authentication and access to our Azure resources: we can push the Docker images we just built to the ACR instance with just a click. You can find a step-by-step guide on how to do this at <https://code.visualstudio.com/docs/containers/quickstart-container-registries>.

If we look at the integrated terminal windows in VS Code, we will see that the `docker push` command has been executed:

```
> Executing task: docker push <ACR repository name>/sample.
microservice.reservationactor.service:latest <
The push refers to repository [dapracrdb.azurecr.io/sample.
microservice.reservationactor.service]
d513a9a5f622: Layer already exists
afc798cc7710: Layer already exists
049b0fdaa27c: Layer already exists
87e08e237115: Layer already exists
```

```
1915427dc1a4: Layer already exists
8a939c4fd477: Layer already exists
d0fe97fa8b8c: Layer already exists
latest: digest: sha256:c6ff3b4058c98728318429ca49f0f8df0
635f7efdfc34b6ccc7624fc3cea7d1e size: 1792
```

Some commands issued by the VS Code extension haven't been shown in the preceding output for brevity. With `docker tag`, an alias has been assigned that matches the ACR address. So, here, our locally built `sample.microservice.reservationactor.service` can be referred to as a combination of <ACR repository name>/<Docker image name><image name>:<tag>.

Each of the images we've built for the Dapr applications should now be pushed to the remote ACR instance.

Once all the container images are available in the registry, we can deploy the Dapr applications to Kubernetes. Before that, we need to connect ACR to AKS.

From a terminal window, we can launch the following command with the Azure CLI, given we have set values for the AKS cluster, `$aksname`, the resource group, `$rg`, and the name of our ACR in a variable, `$acrname`:

```
PS C:\Repos\practical-dapr\chapter09> az aks update --name
$aksname --resource-group $rg --attach-acr $acrname
```

Because we've launched the command in our Azure login context, it executes it. This means it has the correct access rights on both AKS and ACR.

In the next section, we will learn how to pass secrets to Dapr components in Kubernetes.

## Managing secrets in Kubernetes

Secrets such as passwords, connection strings, and keys should always be kept separate from the rest of the code, as well as the configuration of a solution. This is because they could compromise its security if they're shared inappropriately.

Secret management is another building block of Dapr: integration is possible with many secret stores, such as **Azure Key Vault**, **Hashicorp Vault**, and **Kubernetes** itself. A full list is available at <https://docs.dapr.io/developing-applications/building-blocks/secrets/howto-secrets/>.

A Dapr application can retrieve secrets by invoking the Dapr API, which can be reached by issuing the following command:

```
GET http://localhost:<port>/v1.0/secrets/<vault>/<secret>
```

We can also use secrets to configure Dapr components, as documented at <https://docs.dapr.io/operations/components/component-secrets/>. In our sample solution, we are using an Azure Service Bus for the common pub/sub component and an Azure Cosmos DB for the state store of all the components. These rely on keys and connection strings that we need to keep secure.

At this point, we need a secret store. Instead of creating another Azure resource, we can reach a compromise between complexity and another learning opportunity by adopting the Kubernetes built-in secret store, which is readily available via `kubectl`.

To create a secret via the `kubectl` CLI, we can use the `kubectl create secret` syntax, as described in the `Deploy\deploy-secrets.ps1` file. This is shown in the following code snippet:

```
kubectl create secret generic cosmosdb-secret --from-literal=masterKey='#secret#' --from-literal=url='#secret#'
kubectl create secret generic servicebus-secret --from-literal=connectionString='#secret#'
```

We can use the secrets from the Dapr component's `.yaml` files for this. For instance, the following file at `Deploy\component-state-reservationitem.yaml` is the actor state store component, which is mainly used by the `reservationactor-service` Dapr application:

```
apiVersion: daprio/v1alpha1
kind: Component
metadata:
  name: reservationitemactorstore
  namespace: default
spec:
  type: state.azure.cosmosdb
  version: v1
  metadata:
    - name: url
      secretKeyRef:
        name: cosmosdb-secret
        key: url
    - name: masterKey
      secretKeyRef:
        name: cosmosdb-secret
        key: masterKey
    - name: database
      value: state
```

```
- name: collection
  value: reservationitemactorstate
- name: actorStateStore
  value: "true"
```

As you can see, instead of values being directly written in the `component-state-reservationitem.yaml` file for metadata, we are referencing the key's `url` and `masterKey` components from the secret; that is, `cosmosdb-secret`.

We must reference the secrets from all the Dapr component's configuration files. Once we've done that, we can start deploying our Dapr applications.

## Deploying applications

So far, we have managed to push the Docker images for our Dapr applications to ACR.

As you may recall from the previous chapters, in Dapr's standalone hosting mode, we launched an application with the Dapr CLI, like this:

```
dapr run --app-id "reservationactor-service" --app-port "5004"
--dapr-grpc-port "50040" --dapr-http-port "5014" --components-
path components" -- dotnet run --project ./sample.microservice.
reservationactor.service/sample.microservice.reservationactor.
service.csproj --urls="http://+:5004"
```

In Kubernetes hosting mode, we will deploy the Dapr application with a `.yaml` file, configuring all the parameters present in the previous snippet.

The `Deploy\sample.microservice.reservationactor.yaml` file that corresponds to the `reservationactor-service` Dapr application looks as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reservationactor-service
  namespace: default
  labels:
    app: reservationactor-service
spec:
```

```
replicas: 1
selector:
  matchLabels:
    app: reservationactor-service
template:
  metadata:
    labels:
      app: reservationactor-service
    annotations:
      dapr.io/enabled: "true"
      dapr.io/app-id: "reservationactor-service"
      dapr.io/app-port: "80"
      dapr.io/log-level: "info"
  spec:
    containers:
      - name: reservationactor-service
        image: <registry>/sample.microservice.
          reservationactor:2.0
        ports:
          - containerPort: 80
        imagePullPolicy: Always
```

The previous snippet is a standard configuration file for a Kubernetes deployment. `<registry>/sample.microservice.reservationactor`, where `<registry>` must be replaced with the ACR you provisioned previously, specifies the use of a container image with the `2.0` tag. The Dapr-specific portion is all in the metadata annotations: the `dapr.io/enabled: "true"` annotation informs the Dapr control plane operating in Kubernetes that the Dapr sidecar container must be injected into this Pod, granting it the ability to access the Dapr building blocks. The `dapr.io/id: "reservationactor-service"` annotation specifies the Dapr application's unique ID.

Many other Dapr annotations can be used in Kubernetes to influence deployments. For an exhaustive list, please check the documentation at <https://docs.dapr.io/reference/arguments-annotations-overview/>.

This is the overall, and minimal, impact Dapr has on the deployment configuration of microservices destined to be deployed on Kubernetes.

**Docker Hub ready-to-use images**

Ready-to-use container images for each of the sample Dapr applications are available on Docker Hub, and the corresponding `.yaml` files and scripts are available in this chapter's working area. As an example, the `Deploy\sample.microservice.reservationactor.dockerhub.yaml` file contains the deployment that corresponds to the one we just analyzed.

Armed with the necessary components and deployment files, we are ready to deploy the solution to Kubernetes.

To make this step simpler, two files have been provided for you in `C:\Repos\dapr-samples\chapter09`:

- The `Deploy\deploy-solution.ps1` file can be used if you are leveraging your own ACR to keep the application container images safe. Before using it, remember to replace the variables, in the referenced files, with the values that apply to your situation.
- The `Deploy\deploy-solution.dockerhub.ps1` file is leveraging the images on Docker Hub.

As an example, looking at the `Deploy\deploy-solution.ps1` file, the commands are meant to be executed from the solution's root folder, which in my case is `C:\Repos\dapr-samples\chapter09`:

```
kubectl apply -f .\Deploy\component-pubsub.yaml --namespace $namespace
kubectl apply -f .\Deploy\sample.microservice.order.yaml --namespace $namespace
```

The first command applies the configuration of a Dapr component to a namespace in our Kubernetes cluster. The second command applies the deployment manifest for a Dapr application to the same namespace.

We can check the impact that the new deployment has on Kubernetes from the Azure portal via `kubectl` or the Dapr dashboard. With `dapr dashboard -k`, we can see that all our Dapr applications have been configured in Kubernetes, as shown in the following screenshot:

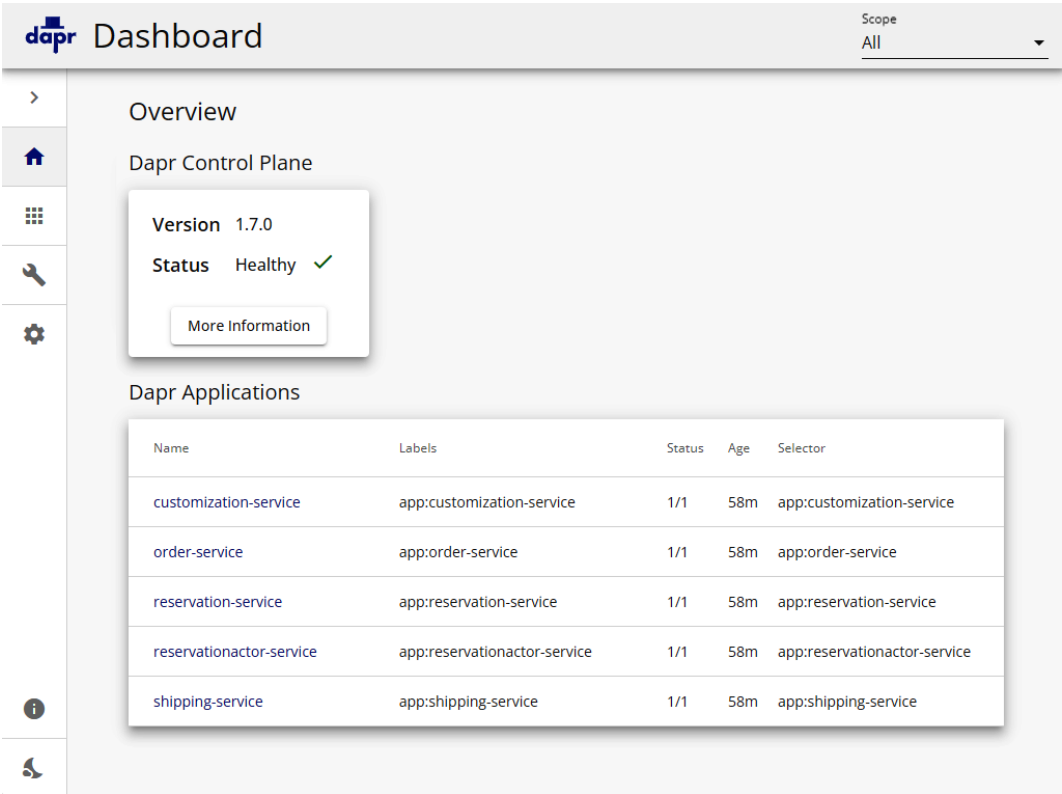
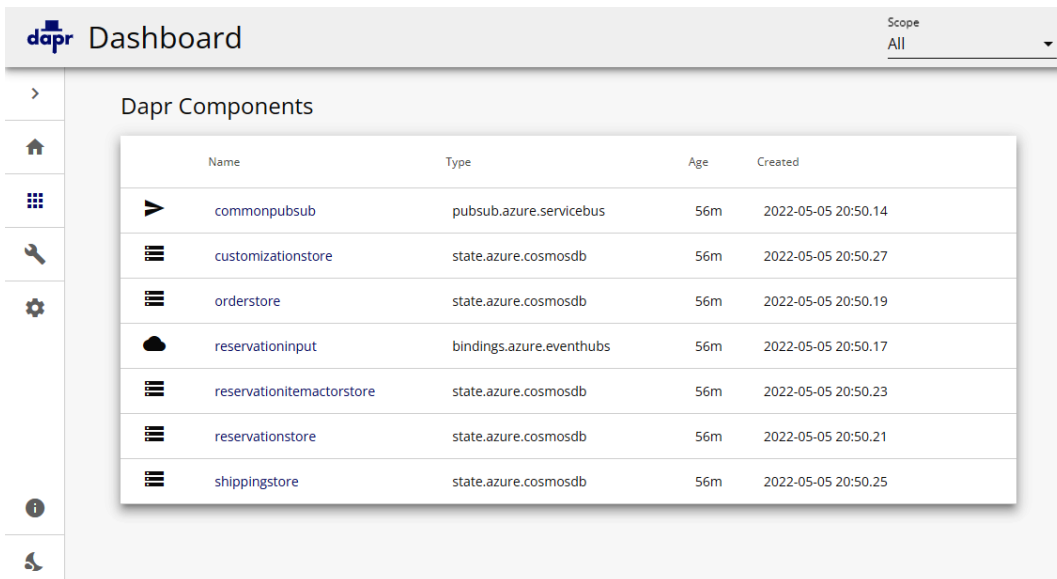


Figure 9.4 – The Dapr Applications view in the Dapr dashboard

Here, we can see the list of Dapr applications that are available, shown by their Dapr application IDs. In the following screenshot, we can see the **Dapr Components** view in the Dapr dashboard:



Name	Type	Age	Created
commonpubsub	pubsub.azure.servicebus	56m	2022-05-05 20:50.14
customizationstore	state.azure.cosmosdb	56m	2022-05-05 20:50.27
orderstore	state.azure.cosmosdb	56m	2022-05-05 20:50.19
reservationinput	bindings.azure.eventhubs	56m	2022-05-05 20:50.17
reservationitemactorstore	state.azure.cosmosdb	56m	2022-05-05 20:50.23
reservationstore	state.azure.cosmosdb	56m	2022-05-05 20:50.21
shippingstore	state.azure.cosmosdb	56m	2022-05-05 20:50.25

Figure 9.5 – The Dapr Components view in the Dapr dashboard

Before we move on to the next section, let's verify the output of the Dapr application.

By using the `kubectl logs -f` command, we can follow the output log from any app and container, in a similar fashion to what we did in the local development environment via the Dapr CLI:

```
PS C:\Repos\dapr-samples\chapter09> kubectl logs -l
app=reservationactor-service -c reservationactor-service
--namespace default -f
info: Microsoft.Hosting.Lifetime[14]
    Now listening on: http://[::]:80
info: Microsoft.Hosting.Lifetime[0]
    Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
    Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
    Content root path: /app/
```

The preceding output comes from the ASP.NET containers in the Pods for the deployment that corresponds to the `reservationactor-service` Dapr application: for the time being, we can only see log events related to its startup.

## Exposing Dapr applications to external clients

Our objective is to expose the ASP.NET endpoints of the Dapr applications, starting with `order-service`, so that we can invoke the `/order` API method from our client machine. The following diagram shows what we are trying to achieve:



In the preceding diagram, the main Dapr services are depicted in Kubernetes alongside our Dapr applications. They are represented as Pods containing the ASP.NET container, along with the service code and the Dapr sidecar.

We need to configure our Kubernetes cluster with an **ingress controller (IC)**. For this, we can use **NGINX**. A detailed step-by-step configuration is available in the Azure documentation at <https://docs.microsoft.com/en-us/azure/aks/ingress-basic>.

For your convenience, the `Deploy\deploy-nginx.ps1` file contains all the instructions for preparing the **NGINX IC** and the ingresses according to our sample solution. The first step is to install NGINX via **Helm**:

```
helm install nginx-ingress ingress-nginx/ingress-nginx `
  --namespace $namespace `
  --set controller.replicaCount=2 `
```

Once the deployment has been completed, we can verify that the IC is ready with the following command:

```
PS C:\Repos\dapr-samples\chapter09> kubectl --namespace default
get services -o wide -w nginx-ingress-ingress-nginx-controller
NAME          TYPE          CLUSTER-IP  EXTERNAL-IP  PORT(S)          AGE    SELECTOR
nginx-ingress-ingress-nginx-controller  LoadBalancer  10.0.21.122  x.y.z.k      80:32732/TCP,443:30046/TCP  41h    app.kubernetes.io/component=controller,app.kubernetes.io/instance=nginx-ingress,app.kubernetes.io/name=ingress-nginx
```

Let's take note of the public IP (the `x.y.z.k` address) of the IC from the `EXTERNAL-IP` column.

Once the IC has been deployed, we must configure an **ingress** that corresponds to each **service** we want to expose.

By examining the `Deploy\services.yaml` configuration file, we can observe the configuration of resources of the (kind) Service type for `order-service`, followed by `reservation-service`:

```
apiVersion: v1
kind: Service
metadata:
  name: order-service
  namespace: default
spec:
  type: ClusterIP
```

```
ports:
- port: 80
selector:
  app: order-service
---
... omitted ...
```

The ingresses for `order-service` and `reservation-service` are available in the `Deploy\ingress-nginx.yaml` file, as shown here:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: order-service-ingress
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /$1/$2
spec:
  ingressClassName: nginx
  rules:
  - http:
      paths:
      - path: /bbmb/(order)/(.*)
        pathType: Prefix
        backend:
          service:
            name: order-service
            port:
              number: 80
  ---
... omitted ...
```

In the previous configuration snippet, we can see that a resource of the `Ingress` type has been defined. In the `spec` field, we can find everything related to **NGINX**: a path is defined to identify requests to route to the backend service named `order-service`. A similar ingress is defined to reach the `reservation-service` service.

To apply these configurations to our Kubernetes environment, we must use the following command:

```
kubectl apply -f .\Deploy\services.yaml
kubectl apply -f .\Deploy\ingress-nginx.yaml
```

At this point, we should get an output that looks similar to the following:

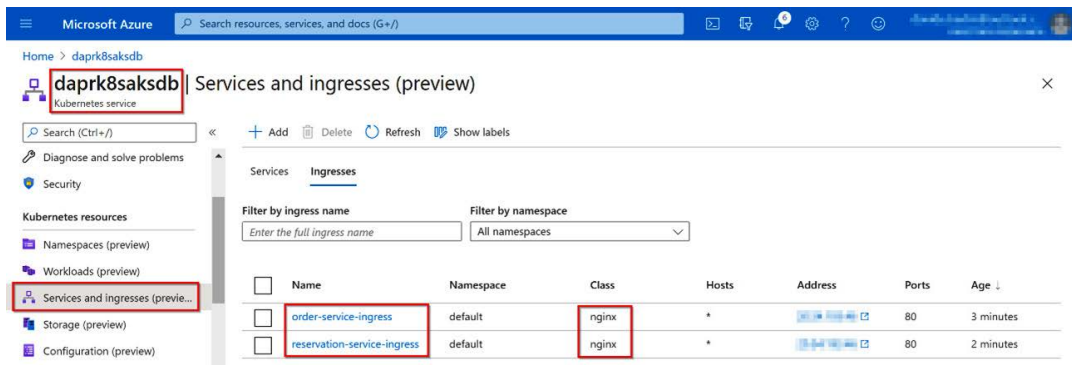


Figure 9.7 – AKS ingress resources view

Here, we can see the two ingresses of the `nginx` class: `order-service-ingress` and `reservation-service-ingress`. While I obfuscated the Address details of my deployment, it is the same for the IC we obtained previously.

We can run a manual test by invoking the HTTP endpoint via cURL or the browser and following the `http://<FQDN or IP>/bbmb/order/` path to reach, via the `order-service` service, the corresponding ASP.NET application. The same applies to `http://<FQDN or IP>/bbmb/balance/` for the `reservation-service` service.

Even better, by leveraging VS Code extension and the tests prepared in the `order.test.http` file, we can retrieve the balance for a cookie and send an order, as shown in the following screenshot:

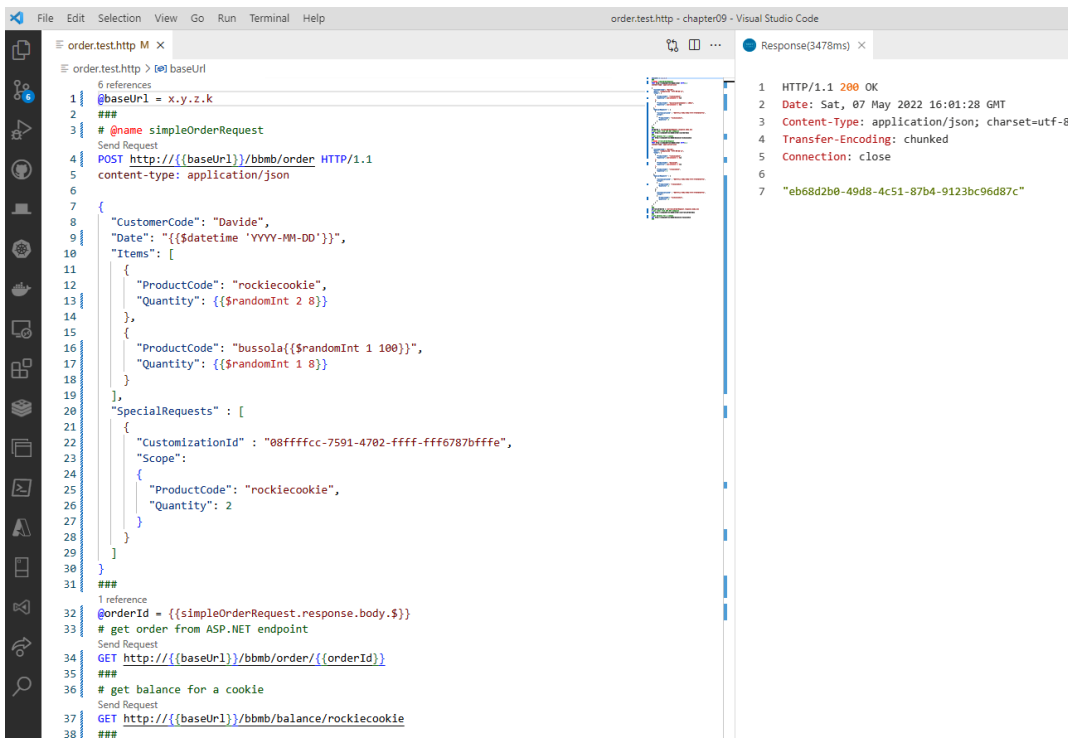


Figure 9.8 – Manual tests in VS Code

Again, if we retrieve the log from the reservationactor-service Dapr application after we invoke it a few times, we should see more logs than we did previously:

```

PS C:\Repos\dapr-samples\chapter09> kubectl logs -l
app=reservationactor-service -c reservationactor-service -
namespace default -f
Activating actor id: rockiecookie
Balance of rockiecookie was 587, now 584
Activating actor id: bussola86
Balance of bussola86 was -3, now -5
  
```

Here, we can see the Dapr virtual actors being activated, and their balance quantity being updated. This is the result of an order that was submitted to the order-service application, which started the saga orchestration we described in *Chapter 6, Publish and Subscribe*, helping it reach the reservationactor-service application we introduced in *Chapter 8, Actors*.

With our IC and ingress configured, we have gained access to the ASP.NET endpoints from outside the AKS cluster, even though we had to configure the services for each Dapr application we wish to expose.

## Summary

This book introduced you to the fictitious e-commerce site *Biscotti Brutti Ma Buoni*. In the previous chapters, we built prototypes for several microservices. Along the way, we learned how the building blocks of Dapr enable any developer, using any language on any platform, to accelerate the development and deployment of a microservice architecture.

We have stayed focused on the building blocks of Dapr and how to combine them optimally, always remaining in the context of the local development environment. We did this by relying on Dapr's standalone mode to test and debug our microservice code.

In this chapter, we finally shifted gears and moved toward a production-ready environment for our Dapr applications, such as a Kubernetes cluster. We learned how to configure Dapr on a Kubernetes cluster, as well as how to handle secrets and components, deploy applications, and configure ingress.

While we managed to expose our ASP.NET applications to external clients, it is important to note that we accomplished this with a generic Kubernetes approach, without leveraging any of the capabilities of Dapr.

In the next chapter, we will learn about a more interesting approach to integrating Dapr with ingress controllers in Kubernetes and how to integrate an API gateway with Dapr.

# Exposing Dapr Applications

In this chapter, we will focus on how to expose our **Distributed Application Runtime (Dapr)** applications hosted in a **Kubernetes Cloud Native Computing Foundation (CNCF)**-compliant cluster, as it is one of the preferred hosting modes for the production environment in Dapr.

In the previous chapter, we managed to expose our **ASP.NET** API in the simplest possible way. Using it as a starting point, we will now learn how to expose our microservices running in the **Azure Kubernetes Service (AKS)** cluster with a different approach that fully leverages Dapr.

In this chapter, we will cover the following topics:

- Daprizing ingress controllers
- Setting up API management on Kubernetes
- Configuring Dapr policies with API management

Continuing in our effort to shift from development to the deployment phase of our journey in microservices architecture, it is important to understand the implications of the different options to expose our APIs to clients and how these can influence our design choices, as well as impact our overall solution architecture.

Our first objective is to change our approach to ingress controllers in the Kubernetes cluster to fully leverage the Dapr capabilities.

## Technical requirements

In this chapter, the working area for the scripts and code can be found at `<repository path>\chapter010\`. In my local environment, it can be found here: `C:\Repos\dapr-samples\chapter10`.

Please refer to the *Setting up Dapr* section of *Chapter 1, Introducing Dapr*, for a complete guide on the tools needed to develop with Dapr and work with the examples provided.

## Daprizing ingress controllers

Let's recap the current stage of our deployment of the *Biscotti Brutti Ma Buoni* backend solution to Kubernetes: after we followed all the steps in *Chapter 9, Deployment to Kubernetes*, we have the Dapr components and applications properly configured, and external calls can reach the APIs of some of our services.

Considering the approach that we followed in exposing the Dapr applications so far, how different has it been from exposing any other generic ASP.NET APIs? Not different at all: taking Dapr out of the picture, we still had to deal with Kubernetes-specific concepts such as services and then set these **services** as the backend of **ingress** rules using the **NGINX Ingress Controller (IC)** we set up.

As you might already guess from the direction the chapter is aiming toward, there are other approaches to inject Dapr into any existing application leveraging HTTP routes and, as an extension of this, an NGINX IC. We are now going to Daprize the NGINX deployment of the ingress controller.

Let's examine the following diagram:

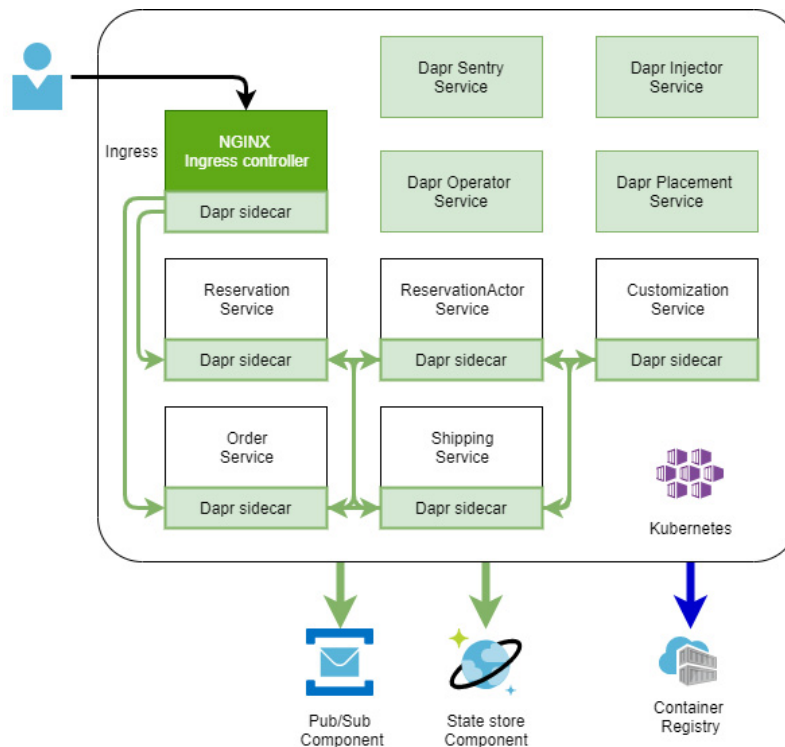


Figure 10.1 – A Daprized IC with NGINX in a Kubernetes deployment

In *Figure 10.1*, we can see that the IC is a proper Dapr application, with its local **Dapr sidecar** and, consequently, with access to any other Dapr application of our solution, without the need to configure Kubernetes services for any of the ASP.NET microservices.

Before we delve into the instructions, let's reflect on this approach, as it shows us the Dapr potential in a wider range of scenarios than NGINX reverse proxies used to expose services in the Kubernetes cluster.

In this book, we learned how to create new ASP.NET microservices and, leveraging the Dapr SDK for .NET, we understood how to programmatically interact with Dapr. By often testing our Dapr applications using the Dapr sidecar URLs, we also verified that direct interaction with the Dapr APIs is always a possibility.

In an existing application, or in a situation in which we have little to no ability to intervene in the application code, we are able to introduce Dapr if we can influence the HTTP endpoint and interaction steps. Luckily for us, we can achieve this in NGINX: we can configure the backend and rewrite the URLs of the ingress rules.

In our AKS cluster, we should already have installed an NGINX IC with the default configuration and named it `nginx-ingress`. To simplify the instructions in the chapter, I suggest uninstalling it first with the following command:

```
PS C:\Repos\practical-dapr\chapter10> helm uninstall nginx-  
ingress release "nginx-ingress" uninstalled
```

With the following configuration, which is available in the `Deploy\nginx-dapr-annotations.yaml` file, we can influence the deployment of the NGINX IC with Dapr-specific annotations we know and well recognize:

```
controller:  
  podAnnotations:  
    dapr.io/enabled: "true"  
    dapr.io/app-id: "nginx-ingress"  
    dapr.io/app-protocol: "http"  
    dapr.io/app-port: "80"
```

The previous configuration file is used to influence the NGINX IC deployment, as shown here:

```
helm install nginx-ingress ingress-nginx/ingress-nginx `  
  --namespace default `br/>  -f .\Deploy\nginx-dapr-annotations.yaml `br/>  --set controller.replicaCount=1
```

For your convenience, the `Deploy\deploy-nginx-dapr.ps1` file contains all the steps, charted with **Helm**, for preparing the NGINX IC with Dapr and the ingresses according to our sample solution.

Let's take note of `EXTERNAL-IP` of the ingress controller with the following command:

```
PS C:\Repos\dapr-samples\chapter10> kubectl --namespace default
get services -o wide -w nginx-ingress-ingress-nginx-controller
NAME      TYPE      CLUSTER-IP  EXTERNAL-IP  PORT(S)  AGE   SELECTOR
nginx-ingress-ingress-nginx-controller  LoadBalancer
10.0.27.161  x.y.z.k    80:32732/TCP,443:30046/TCP  1h
app.kubernetes.io/component=controller,app.kubernetes.io/
instance=nginx-ingress,app.kubernetes.io/name=ingress-nginx
```

As we first uninstalled and later reinstalled NGINX, the chances are the IP would be changed.

Once deployed, the Pods of the NGINX IC will be injected with the Dapr sidecar container. This change enables us to configure Ingress without the need for additional Kubernetes services. Let's see the ingress rules in the `Deploy\ingress-nginx-dapr.yaml` file:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: order-dapr-ingress
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /v1.0/
invoke/          order-service/method/order/$1
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
          - path: /order/(.*)
            pathType: Prefix
            backend:
              service:
                name: nginx-ingress-dapr
                port:
```

```
        number: 80
---
... omitted ...
```

The main points of attention in the ingress configuration of a Dapr-enabled NGINX IC are `rewrite-target` and `backend`: for the latter, we use the `service` that Dapr automatically creates for each annotated application, which is called `nginx-ingress-dapr` in this case.

With the `rewrite-target` annotation, we can expose an easy-to-use URL externally while rewriting it so that we can leverage the more complex Dapr pattern too.

In this case, we are instructing NGINX to direct all requests for the `/order/` route to the Dapr sidecar, using the service-to-service invocation building block rewriting the path to `/v1.0/invoke/order-service/method/order/`; in our intention, this request will ultimately reach the `order-service` Dapr application. A similar ingress rule is configured for the `/balance/` route to reach the `reservation-service` application.

Let's apply the new ingress rules to our Kubernetes cluster with the following command:

```
PS C:\Repos\practical-dapr\chapter10> kubectl apply -f .\
Deploy\ingress-nginx-dapr.yaml
ingress.networking.k8s.io/order-dapr-ingress created
ingress.networking.k8s.io/reservation-dapr-ingress created
```

We can run our manual tests by invoking the HTTP endpoint via the **Visual Studio Code (VS Code)** extension, following the tests prepared in the `nginx-test.http` file, replacing the base URL with the `EXTERNAL-IP` address we previously collected. With that, we've verified that our new ingresses are working properly: a request to `http://<FQDN or IP>/balance/<SKU>` will reach the `reservation-service` application via Dapr.

With this alternative approach to the ingress controller, compared to the classic one presented in the *Exposing Dapr applications to external clients* section of *Chapter 9, Deployment to Kubernetes*, we have successfully exposed our Dapr applications with less effort as no services have to be configured. It also taught us how to integrate Dapr into an existing application, such as NGINX, to increase its capabilities.

In the next section, we will learn how to integrate Dapr with an API manager such as the **Azure API Management (APIM)** service in the Azure cloud platform.

## Setting up API management on Kubernetes

First of all, why would you consider adding an API management service to your architecture?

In *Chapter 3, Microservices Architecture with Dapr*, we explored many concepts, including the relevance of the contracts we, architects of our solution and therefore of its external API, implicitly sign with our consumers.

To simplify a complex topic, great care should be taken in evolving an API on which other applications and systems depend; the way our overall solution changes over time, in terms of architecture, language, or frameworks, should never impact any external consumer in an unplanned or—even worse—an unexpected way.

As architects and developers, how can we manage the external API so that it does evolve to match objectives and at a different pace than our internal microservices? A great tool at our disposal is an API manager: a powerful way to manage the life cycle of an API, starting from its creation, documenting how to use it, and governing its adoption.

APIM can help us achieve this ambitious goal. It is a **Platform as a Service (PaaS)** in the Azure cloud, made up of three components:

- **Developer portal:** A fully customizable portal to publish rich documentation of our APIs to developers
- **Management plane:** This is the core of the Azure service, offering a set of functionalities to define, configure, and package APIs into a product, govern access via a subscription, and insert policies to influence how APIs are processed and analyzed
- **Gateway:** This is the workhorse of the service, evaluating requests from external clients, applying policies, interacting with backends, and collecting metrics and traces that can be later analyzed

You can find more information in the APIM documentation at <https://docs.microsoft.com/en-us/azure/api-management/api-management-key-concepts>. While APIM is a vast service providing a wide range of features, we will focus on the gateway deployment and the policies we can use in the management plane in the scope of our solution.

The gateway component of an APIM resource can be provisioned in **Self-hosted** mode: the gateways will be deployed as container images in the same environment in which your backend API is operating—for example, in the same Kubernetes cluster as the main production scenario we are considering in this chapter. The self-hosted gateway deployed to Kubernetes will be kept in sync with the changes applied via the management plane.

The following diagram provides an overview of this:

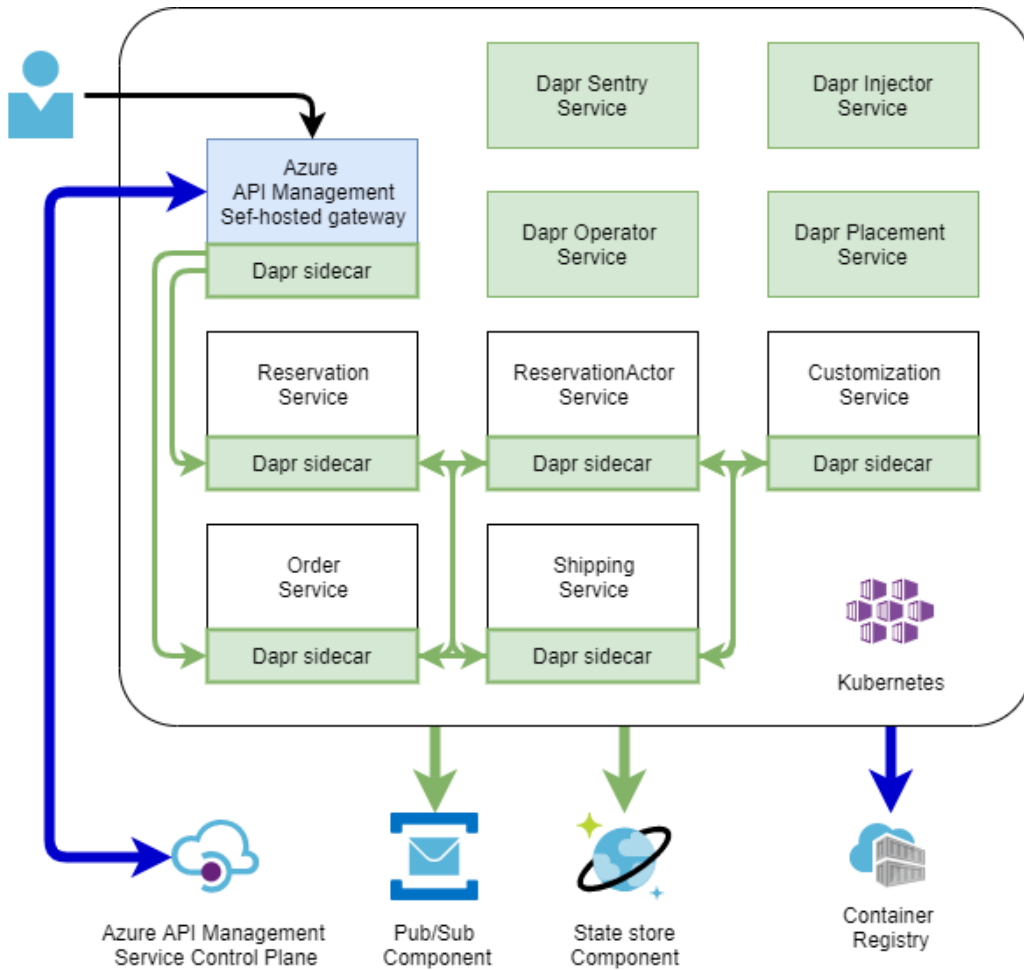


Figure 10.2 – APIM self-hosted mode with Dapr

As depicted in *Figure 10.2*, we can see how this deployment does look like the Daprized NGINX deployment we explored in the previous section (*Figure 10.1*). It is important to note that while NGINX is just a reverse proxy, APIM is a fully fledged API gateway.

Before continuing with the chapter, please follow the instructions to create an APIM service via the portal, as described in the documentation at <https://docs.microsoft.com/en-us/azure/api-management/get-started-create-service-instance>, or with any other approach; there are no particular settings to specify at this stage.

Once we have provisioned the APIM service, we are ready to deploy the gateway. You can follow detailed instructions here: <https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-provision-self-hosted-gateway>. The following screenshot shows the starting point for deploying the gateway in the same Kubernetes cluster hosting our solution:

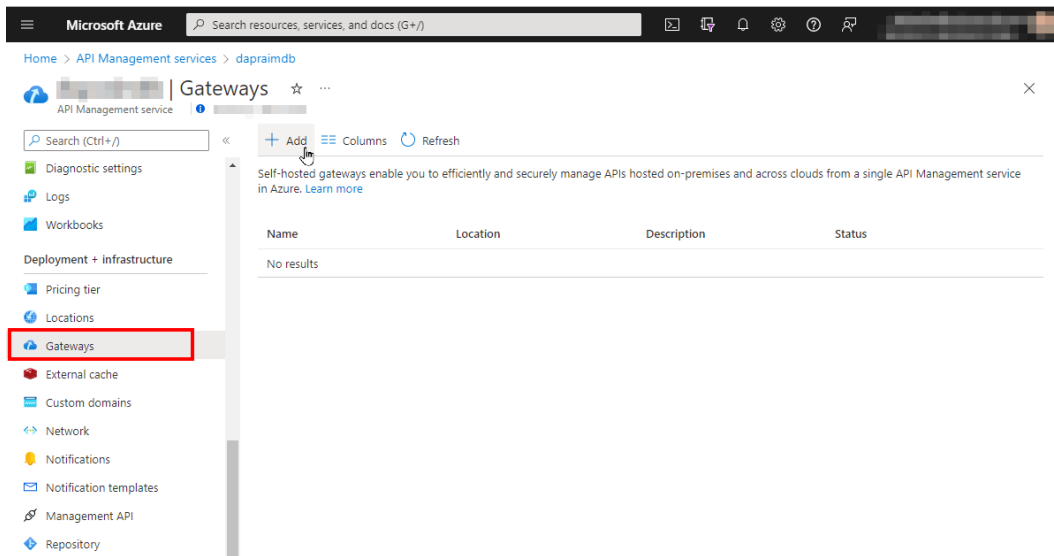


Figure 10.3 – Adding gateway to APIM

Let's name our gateway as we prefer. Once defined, we can examine the Kubernetes configuration for the self-hosted gateway. As depicted in the following screenshot, APIM does provide a way to configure each gateway:

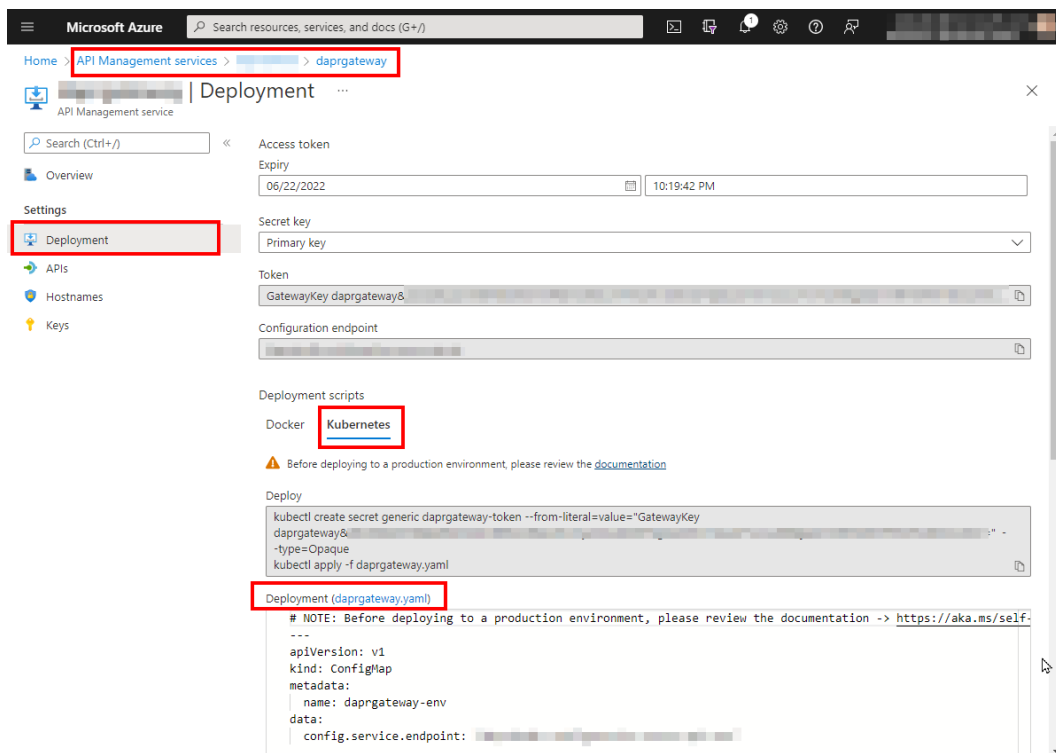


Figure 10.4 – Deployment settings for APIM

The Azure portal, as shown in *Figure 10.4*, provides a default deployment manifest for our specific self-hosted gateway. I suggest you copy the content or download the `.yaml` file, which we need to edit before we deploy it to Kubernetes.

There are `ConfigMap`, `Deployment`, and `Service` objects defined in the file. The changes we need to apply to the `.yaml` to enable Dapr are minimal: in the `Template` section of the `Deployment` object, we add annotations for Dapr, as in the following snippet:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: daprgateway
... omitted ...
template:
  metadata:
    labels:
```

```

    app: daprgateway
    annotations:
      dapr.io/enabled: "true"
      dapr.io/app-id: "apim"
  spec:
... omitted ...

```

In my example, I assigned a Dapr application name of `apim` to the gateway deployment named `daprgateway`. This step is very well described in the Dapr documentation for APIM, available at <https://docs.microsoft.com/en-us/azure/api-management/api-management-dapr-policies#enable-dapr-support-in-the-self-hosted-gateway>.

We are now ready to first create a secret provided for the gateway, as shown in the portal in *Figure 10.4*. Here's how we can do this:

```

PS C:\Repos\practical-dapr\chapter10> kubectl create secret
generic daprgateway-token --from-literal=value="GatewayKey
..." --type=Opaque
secret/daprgateway-token created

```

Next, let's deploy the self-hosted gateway annotated with Dapr. In my case, I saved and edited the file with the name `Deploy\daprgateway-dapr.yaml`, as shown here:

```

PS C:\Repos\practical-dapr\chapter10> kubectl apply -f .\
Deploy\daprgateway-dapr.yaml
configmap/daprgateway-env created
deployment.apps/daprgateway created
service/daprgateway created

```

How can we verify whether deployment succeeded and whether Dapr is enabled?

I named my gateway `daprgateway`; therefore, the corresponding service should have the same name. We can see here that is the case:

```

PS C:\Repos\practical-dapr\chapter10> kubectl get services
daprgateway

```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
daprgateway			

```

PORT(S)
AGE

```

```
daprgateway    LoadBalancer    10.0.9.161    x.y.z.k
80:32232/TCP,443:32501/TCP    12m
```

EXTERNAL-IP is how we can reach the APIM self-hosted gateway running in our Kubernetes cluster from external clients.

Does the gateway appear as a Dapr application too? Let's check with the following command:

```
PS C:\Repos\practical-dapr\chapter10> dapr list -k NAMESPACE
APP ID          APP PORT  AGE    ... omitted ...
default         apim      16m    ...
default         nginx-ingress  80 8d    ...
... omitted ...
```

Among the Dapr applications corresponding to our microservices, there should be the NGINX IC and the Dapr-enabled, self-hosted gateway named, in my case, `apim`.

We successfully provisioned the APIM service and deployed a self-hosted gateway to our previously arranged AKS cluster, not before changing the configuration to make it a fully functional Dapr application.

As we are now confident the endpoint of our API manager is ready to receive external client requests, the next step is to explore how APIM supports Dapr with specific policies.

## Configuring Dapr policies with API management

To reach our goal, we have to create an API first and then a set of **operations**, not forgetting to deploy the API to our **self-hosted gateway**.

To have a good understanding of the terminology of APIM, I suggest reading the documentation at <https://docs.microsoft.com/en-us/azure/api-management/api-management-terminology>.

Let's create an API first, as shown in the following screenshot:

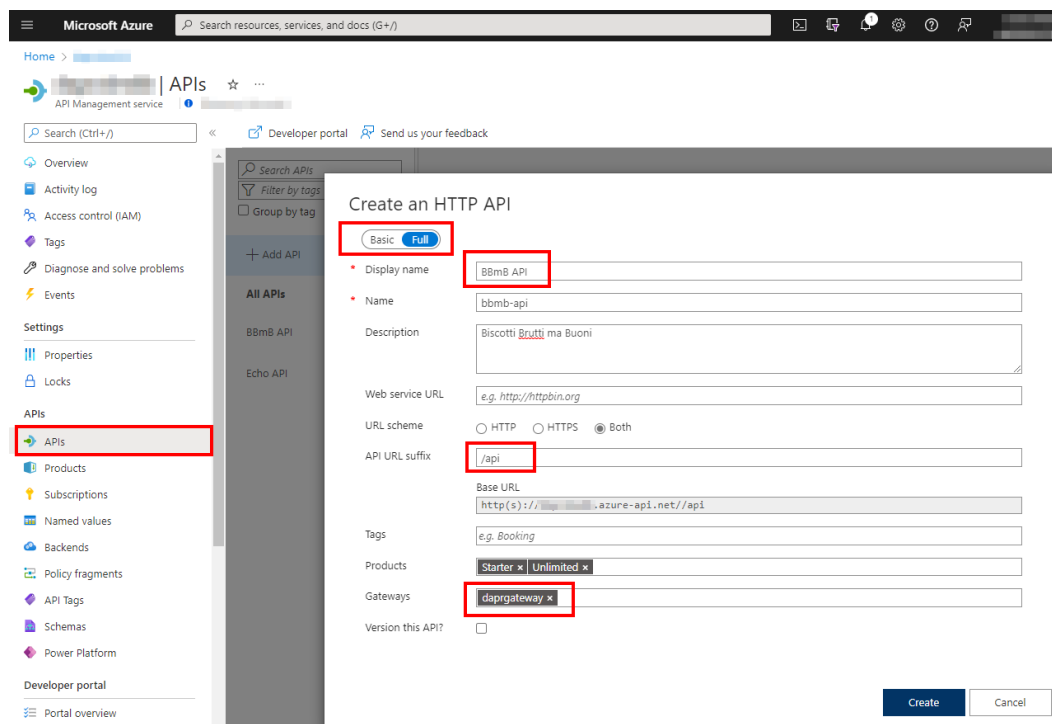


Figure 10.5 – Creating an API

As depicted in *Figure 10.5*, I chose to configure the API; the API named BBmB API has a `/api` URL suffix. It is present in all the default products (this is just to simplify the scenario), and I would like to deploy it only to the self-hosted gateway named `daprgateway` we created in the previous section.

With an API defined, we are now ready to create our first operation to check whether everything works as expected. The following screenshot highlights how to do this:

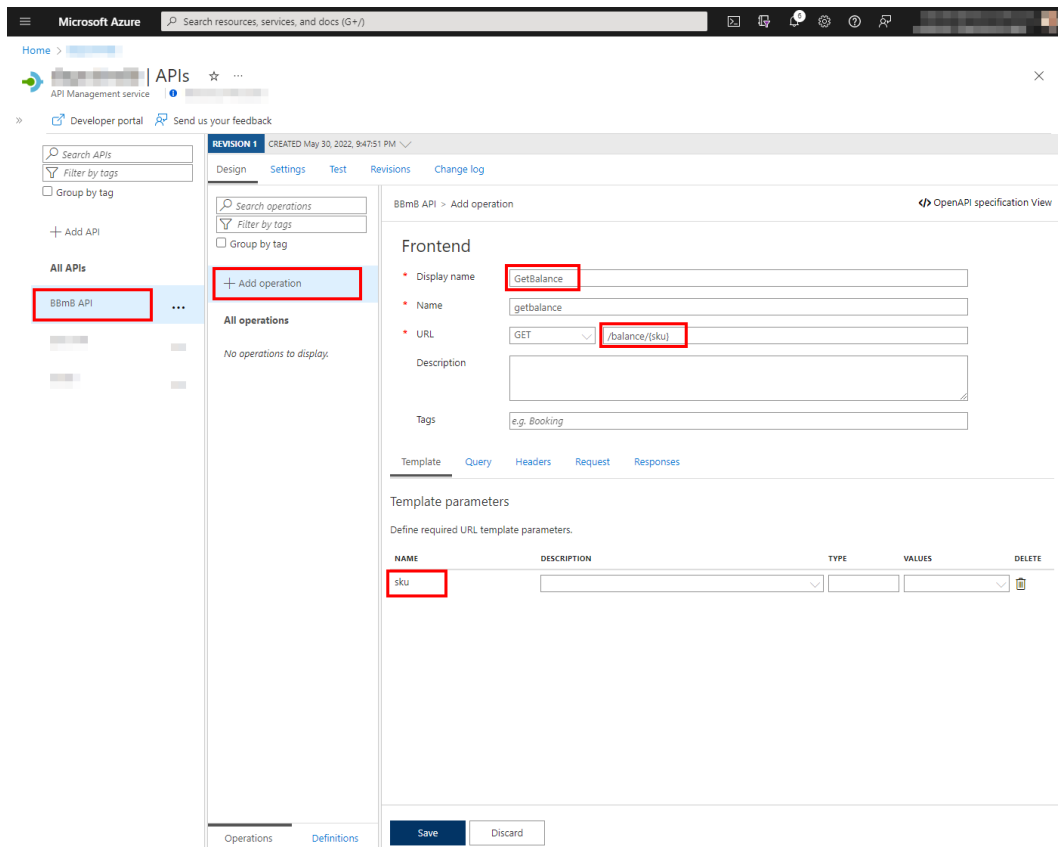


Figure 10.6 – Creating an operation

As shown in *Figure 10.6*, we select the API we just created to add an operation. I named the operation `GetBalance`, specifying the URL matching this GET operation to be `/balance/{sku}` where `{sku}` is a template parameter we can further leverage in operation.

Once we save its definition, we can configure the operation. We will intervene only in the **Inbound processing** policies, as shown in the next screenshot:

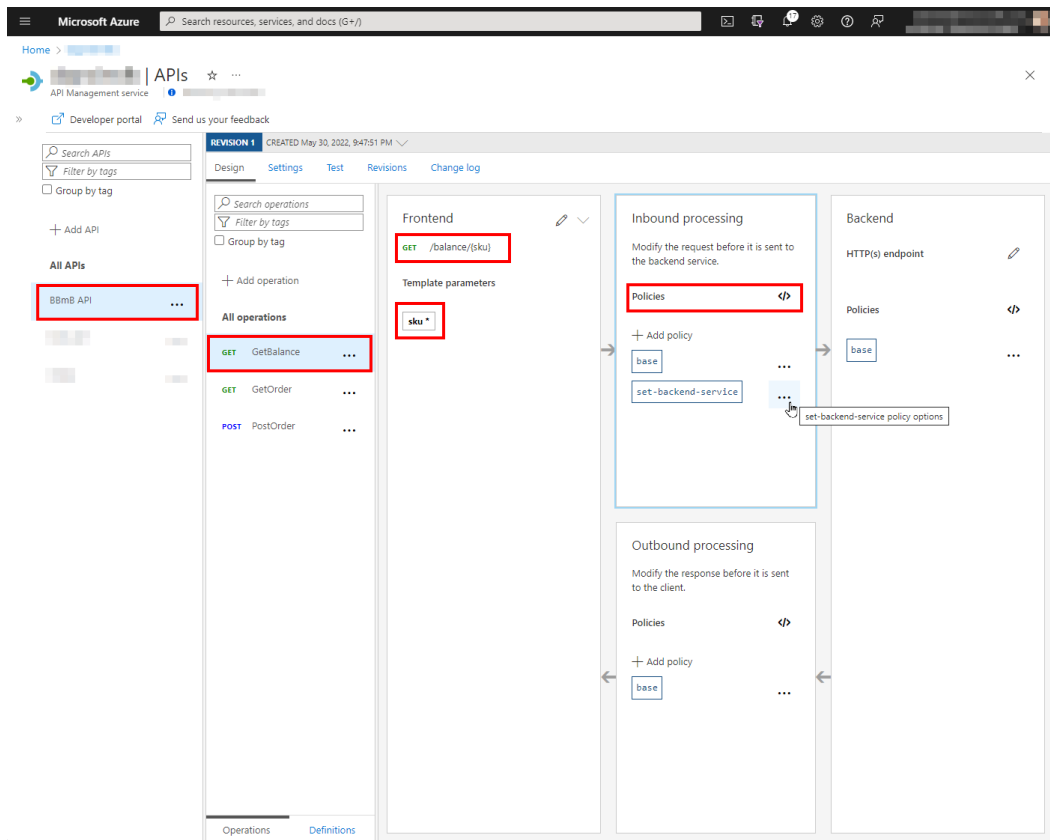


Figure 10.7 – Configuring policies

Before we explore the details of our operation policies, let's recap our objective. We want to define API operations to route all matching requests, reaching the self-hosted gateway deployed to our Kubernetes cluster, to the Dapr applications of our choice.

As we enabled Dapr in the self-hosted gateway, we can leverage the three Dapr-integrated policies provided by APIM. You can learn all about this in the documentation at <https://docs.microsoft.com/en-us/azure/api-management/api-management-dapr-policies>. To achieve our goal, we will leverage the service invocation policy, while I leave the **publish/subscribe (pub/sub)** and outbound binding policies to your experimentations.

The following snippet, which you can find in the `API-policies\apimpolicies-getbalance.xml` file, shows the **XML** corresponding to all policies of our operation. Let's focus on the inbound processing section:

```
<policies>
  <inbound>
```

```

    <base />
    <set-backend-service backend-id="dapr" dapr-app-
      id="reservation-service" dapr-method="@
    (String.Format("balance/
      {0}", context.Request.
    MatchedParameters["sku"]))" dapr-
    namespace="default" />
  </inbound>
  <backend>
    <base />
  </backend>
  <outbound>
    <base />
  </outbound>
  <on-error>
    <base />
  </on-error>
</policies>

```

With the `set-backend-service` policy, we redirect the requests to a specific `backend-id`; as the `dapr` value suggests, it is going to be the Dapr sidecar. This is the attribute to enable the Dapr integration in our API operation.

With `dapr-app-id`, we set the Dapr application we want to invoke via a service-to-service invocation building block—the `reservation-service` application in this case.

In creating the API operation, we specified a `{sku}` template parameter. In the `dapr-method` attribute, we leverage a slightly more complex approach, by using a policy expression in `C#` to combine the `balance` method with the value of the matching `{sku}` parameter.

We can save the policy definition: the APIM control plane will immediately propagate the policy to the self-hosted gateway deployed to the Kubernetes cluster.

If we managed to correctly configure our API operation, a request to `https://<EXTERNAL-IP>/api/balance/rookiecookie`, where `EXTERNAL-IP` is the IP of the self-hosted gateway, will be translated to `http://localhost:3500/v1.0/invoke/order-service/method/balance/rookiecookie`, reaching the Dapr application and returning the result back to the client.

### Subscriptions and Products in APIM

In APIM, a subscription is the ability, via a pair of subscription keys, to access all of the API or the APIs included in a product.

If you followed the instructions in this chapter, you configured the API to require a subscription key to gain access to it. In this case, you need a key to pass in the `Ocp-Apim-Subscription-Key` header.

You can follow the instructions on how to obtain a subscription key at <https://docs.microsoft.com/en-us/azure/api-management/api-management-subscriptions>.

In the `apim.test.http` test file, I prepared the same request to post an order and get the balance we are used to (before using it, remember to replace the variables). To check the balance of a **SKU** (**stock-keeping unit**), the code of a cookie, we can submit a request composed as follows:

```
###
# get balance for a cookie
GET https://{baseUrl}/balance/rockiecookie
Ocp-Apim-Trace: True
Ocp-Apim-Subscription-Key: {{subscriptionId}}
```

In these headers, I pass a subscription key to obtain access to the API and I demand a traceback from the API showing the **end-to-end (E2E)** processing of the request. I strongly suggest doing this as it is very helpful to understand how the policies work.

If all our work with Dapr integration in the APIM service has been carried out correctly, we should get a positive 200 OK result back from the API, receiving the current balance of the cookie.

To complete the overall scenario, we should configure the operations for submitting an order and retrieving it, both interacting with the `order-service` Dapr application:

- For submitting an order, create an operation named `PostOrder` with a URL of `/order` and a `POST` verb. The policies should be from the `API-policies\apimpolicies-postorder.xml` file:

```
<policies>
  <inbound>
    <base />
    <set-backend-service backend-id="dapr" dapr-app-id="order-service" dapr-method="order" />
  </inbound>
  ... omitted ...
</policies>
```

- For retrieving an order, create an operation named `GetOrder` with a URL of `/order/{orderId}`, a GET verb, and the following policies, as from the `API-policies\apimpolicies-getorder.xml` file:

```
<policies>
  <inbound>
    <base />
    <set-backend-service backend-id="dapr" dapr-app-
      id="order-service" dapr-method="@(<String.
Format("order/
      {0}", context.Request.
MatchedParameters["orderId"]))"
      dapr-namespace="default" />
    </inbound>
    ... omitted ...
  </policies>
```

Finally, let's learn how we can monitor the processing of our requests.

We can use the `kubectl` CLI to get logs from the containers and Pods involved in the deployment of the self-hosted gateway, as with this command:

```
PS C:\Repos\practical-dapr\chapter10> kubectl get deployments
NAME                                READY    UP-TO-DATE
AVAILABLE    AGE
customization-service              1/1      1
1                27d
daprgateway                        1/1      1
1                9d
nginx-ingress-ingress-nginx-controller 1/1      1
1                17d
order-service                      1/1      1
1                27d
reservation-service                1/1      1
1                27d
reservationactor-service           1/1      1
1                27d
shipping-service                   1/1      1
1                27d
```

In the previous output, we see `daprgateway` is the deployment name, in case we forgot it. With that, let's observe the logs with the following command. In order to make any message appear in the logs, we can perform some test requests from the `apim.test.http` test file:

```
PS C:\Repos\practical-dapr\chapter10> kubectl logs -f --since
1m deployment/daprgateway
... omitted ...
[Info] 2022-06-4T10:32:46.846, isRequestSuccess: True,
totalTime: 13, category: GatewayLogs, callerIpAddress: x.y.z.k
2.124, timeGenerated: 06/04/2022 10:32:46, region: AKS,
correlationId: 4004e90a-423b-4b7e-96d4-e2e00909a24f, method:
GET
, url: https://EXTERNAL-IP/api/balance/rockiecookie,
backendResponseCode: 200, responseCode: 200, responseSize: 680,
cache: none, backendTime: 12, apiId: api, operationId:
getbalance, productId: unlimited, userId: 1,
apimSubscriptionId:
6273a22ede804a0053070002, clientProtocol: HTTP/1.1,
backendProtocol: HTTP/1.1, backendId: dapr, apiRevision: 1,
clientTl
sVersion: 1.2, backendMethod: GET, backendUrl: http://
localhost:3500/v1.0/invoke/reservation-service.default/method/
bala
nce/rockiecookie, correlationId: 4004e90a-423b-4b7e-96d4-
e2e00909a24f
```

There is a lot of text in the log output, but we can notice that a request from our client (the IP address has been masked with `x.y.z.k`) reached the APIM self-hosted gateway (masked too with `EXTERNAL-IP`) with an `https://EXTERNAL-IP/api/balance/rockiecookie` URL path and has been translated to `http://localhost:3500/v1.0/invoke/reservation-service.default/method/balance/rockiecookie` to reach the backend.

If you enjoy using the Dapr dashboard, which you can launch via the `dapr dashboard -k` command, you can get the same information as shown in the next screenshot:

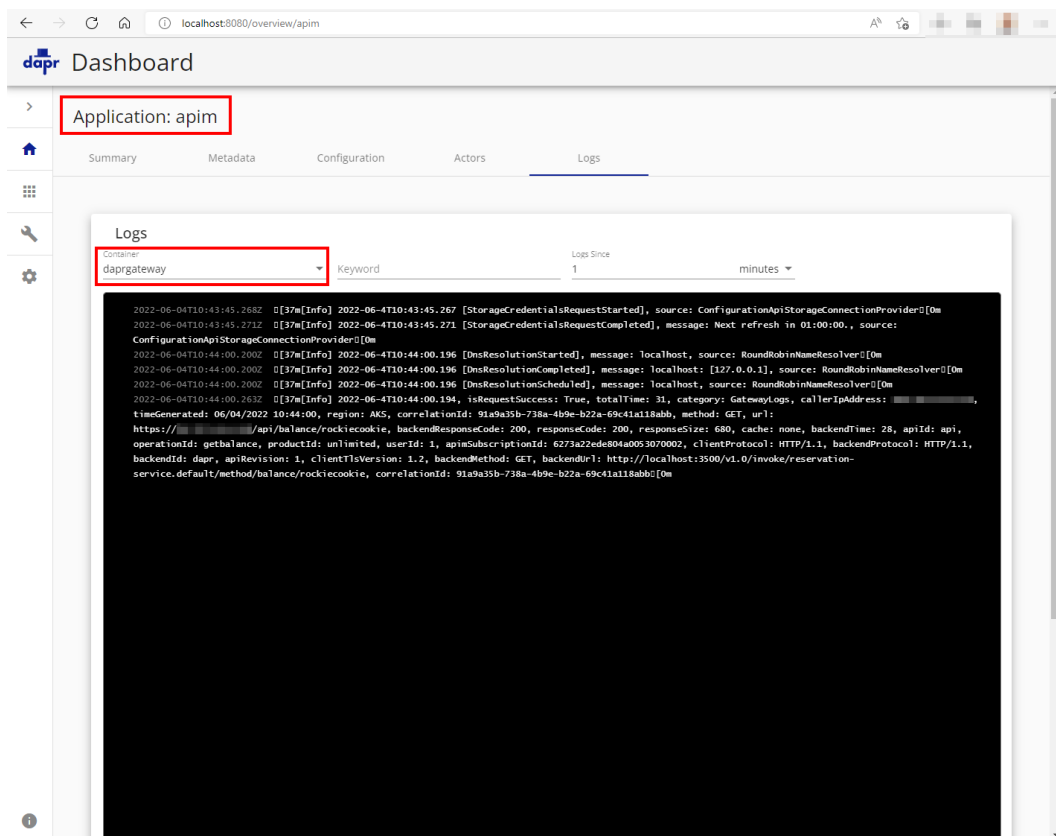


Figure 10.8 – Self-hosted gateway logs via the Dapr dashboard

Another approach is to leverage the AKS cluster **Live Logs** feature on the Azure portal, as described in <https://docs.microsoft.com/en-us/azure/azure-monitor/containers/container-insights-livedata-overview>.

With this, we conclude our exploration of how APIM provides native integration with Dapr thanks to Dapr-integrated policies.

## Summary

So far in this book, we learned how to use the Dapr building blocks to develop microservices. Starting with *Chapter 4* up to *Chapter 8*, we managed to package and deploy our Dapr application to an AKS Kubernetes cluster in Azure in *Chapter 9*.

In this chapter, we learned two different approaches to selectively expose these Dapr applications to clients located externally of the cluster.

First, we experimented with NGINX to understand how to infuse Dapr into an existing application, without any change but instead working on the configuration the reverse proxy relies upon to define ingress rules. In addition to being a valid way to expose the Dapr application, this offers a perspective on how to introduce Dapr into an application without an SDK or any other modification.

Secondly, we explored how APIM can help us define and manage an API exposed to external clients and applications. There is much more to learn about APIM; in this chapter, we focused on the native integration APIM provides for Dapr.

Dapr policies in APIM give architects and developers the ability to leverage the Dapr service-to-service invocation, pub/sub, and resource-binding building blocks directly in the definition of API operations. We used the service-to-service invocation building block in our policies, and I urge you to read more about this powerful integration in the documentation at <https://docs.microsoft.com/en-us/azure/api-management/api-management-dapr-policies>.

In the next chapter, we will learn how, by instrumenting our Dapr applications, we can gain E2E visibility over their usage and performance.

# Tracing Dapr Applications

In this chapter, you will learn about observability options in **Dapr** by exploring how traces, logs, and metrics are emitted and can be collected in Dapr using **Zipkin**, **Prometheus**, and **Grafana**. These are the main topics of the chapter:

- Observing applications in Dapr
- Tracing with Zipkin
- Analyzing metrics with Prometheus and Grafana

With Dapr in self-hosted mode, which we used during the development of our Dapr applications, we had an option to directly access the logs from the Dapr sidecar processes and applications as console output, in addition to the ability to debug our service code with **Visual Studio Code (VS Code)**.

In the Kubernetes mode of Dapr, however, the approach is going to be different because of the complexity of a multi-node cluster and the constraints imposed on a production environment.

## Technical requirements

The code for this sample can be found on GitHub at <https://github.com/PacktPublishing/Practical-Microservices-with-Dapr-and-.NET-Second-Edition/tree/main/chapter11>.

In this chapter, the working area for scripts and code is expected to be `<repository path>\chapter11\`. In my local environment, it is `C:\Repos\dapr-samples\chapter11`.

Please refer to the *Setting up Dapr* section in *Chapter 1, Introducing Dapr*, for a complete guide to the tools needed to develop with Dapr and work with the samples.

This chapter builds upon the Kubernetes cluster we set up and configured in *Chapter 9, Deploying to Kubernetes*; refer to this chapter to reach the same configuration.

## Observing applications in Dapr

In a monolithic application made up of a few components running on a limited number of nodes, understanding how the application is behaving is a relatively simple task. In such a context, there is an expectation that by monitoring the activity of some processes on one or maybe two nodes to get higher availability, and their usage of the nodes' **central processing unit (CPU)** and memory over time, we can get a good perspective on the application's behavior. Log files would be available on the nodes, and the collection of those files could be arranged with any classic monitoring solution.

Once the number of working components increases dramatically, however, when we leverage the advantages coming from building microservices around business capabilities, the level of complexity grows along with it. We have more processes to monitor, and it is also likely that these are related to each other and might even have dependencies on external components; as we learned throughout this book, microservices often collaborate via a service-to-service invocation or **publish/subscribe (pub/sub)** and rely on state stores.

The growth of complexity does not stop here. The life cycle of the microservices composing our solution is shorter; maybe we are able to scale out microservice instances by reacting to the current load.

Our microservice solution will not necessarily need more nodes to run; being more expensive is not an objective! On the other hand, our solution might gain a significant benefit by operating on an infrastructure composed of many self-repairing nodes.

Moreover, the development and deployment practices we learned with Docker give us more flexibility but add an abstraction layer between the node and our code.

We will not understand much of our application's behavior by looking at metrics from any of the nodes.

From this increased (but, alas, necessary and welcome) complexity comes the need for observability of microservices: the ability to instrument new and existing applications to collect metrics, logs, and traces from a diverse set of processes, running on a variable number of hosts (nodes and containers), with the primary objective of correlating each signal with others to offer a complete view of a client request or job.

Let's consider the following simplified representation of the saga pattern, introduced in *Chapter 6, Publish and Subscribe*:

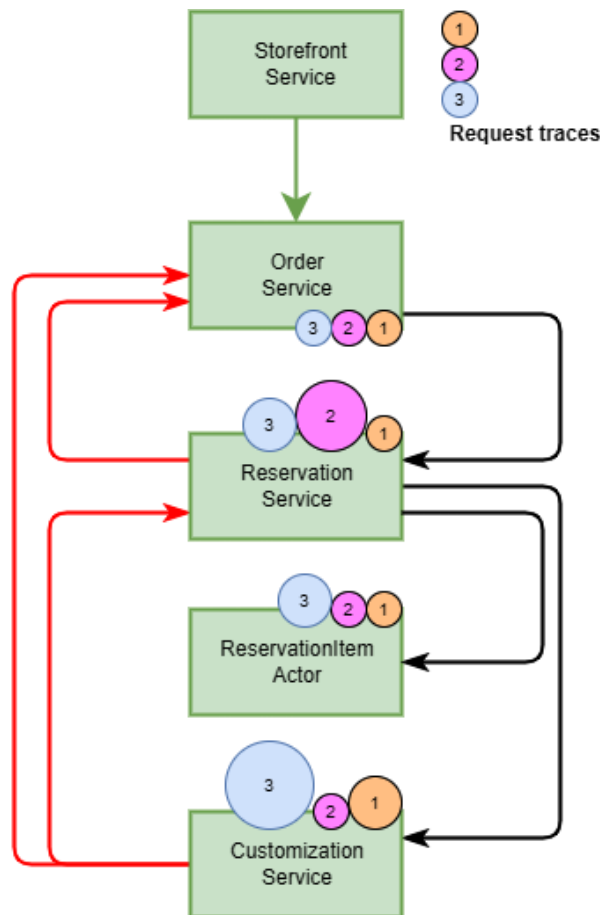


Figure 11.1 – Request traces in a microservice solution

As depicted in *Figure 11.1*, when our solution processes many requests by splitting the work among many microservices, how can we understand which requests are taking more time to execute? How do we account for the interaction between microservices and the state store? To simplify, the goal with observability is to gain a clear view of which request (among 1, 2, or 3) is consuming the most resources.

Dapr supports distributed tracing by automatically correlating traces of a request as it crosses boundaries between the Dapr runtime, our service code, and the Dapr services and components.

As Dapr is a runtime, you only need to configure tracing and decide where to export traces to, nothing more- no frameworks or packages to install and maintain.

Dapr also exposes metrics of the overall Dapr infrastructure, from runtimes in sidecar containers to system services, offering full visibility on how Dapr operates on a Kubernetes (or self-hosted) environment.

**Important note – OpenTelemetry**

Dapr adopts **OpenTelemetry** (see <https://opentelemetry.io/> to learn more), a **Cloud Native Computing Foundation (CNCF)** project with the goal to facilitate integration with frameworks and tools for tracing, metrics, and logs.

With the **OpenTelemetry Collector**, Dapr can export traces to any OpenTelemetry-integrated backend and platforms. The OpenTelemetry Collector offers a vendor-agnostic implementation of how to receive, process, and export telemetry data. You can find more details at <https://docs.dapr.io/operations/monitoring/tracing/open-telemetry-collector/>.

In the next section, we will set up Zipkin to see how distributed tracing can help us understand how our Dapr applications are performing.

## Tracing with Zipkin

**Zipkin** is an open source distributed tracing system. It offers the ability to search for traces by identifier (ID), service, operation, or tags, and shows the dependencies between services. You can learn more at <https://zipkin.io/>.

These are the steps we will follow to set up Zipkin in Dapr on Kubernetes:

1. Setting up Zipkin
2. Configuring tracing with Zipkin
3. Enabling tracing in Dapr
4. Investigating with Zipkin

Let's start by installing Zipkin in the cluster we prepared in *Chapter 9, Deploying to Kubernetes*.

### Setting up Zipkin

Zipkin is distributed as a Docker container. You probably already have it on your local development environment, as it has been installed by default with Dapr.

We can deploy it to Kubernetes with the following `Deploy\deploy-zipkin.yaml` file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: zipkin
  labels:
    app: zipkin
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: zipkin
  template:
    metadata:
      labels:
        app: zipkin
    spec:
      containers:
        - name: zipkin
          image: openzipkin/zipkin
          ports:
            - containerPort: 9411
---
apiVersion: v1
kind: Service
metadata:
  name: zipkin
  namespace: default
  labels:
    app: zipkin
spec:
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 9411
      targetPort: 9411
  selector:
    app: zipkin
```

In the preceding code snippet, Zipkin is deployed with the `openzipkin/zipkin` container exposed at port 9411, and a service endpoint is created with the same port. This is necessary to make it reachable by service name from other Pods.

We can apply the configuration to Kubernetes with the following command:

```
PS C:\Repos\practical-dapr\chapter11> kubectl apply -f .\
Deploy\zipkin.yaml
```

To access Zipkin on Kubernetes, we have two options: as we have an NGINX ingress controller already configured in our cluster, we could create an ingress to the Zipkin service and restrict access to the path from our client **Internet Protocol (IP)** address. Alternatively, we could use the port-forwarding command of `kubectl`, as in the following command:

```
PS C:\Repos\practical-dapr\chapter11> kubectl port-forward svc/
zipkin 9412:9411
Forwarding from 127.0.0.1:9412 -> 9411Forwarding from
[::1]:9412 -> 9411
```

With the `kubectl port-forward` command (see documentation at <https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/>), a local port on our development environment is mapped to a service (this could also be a Pod) on Kubernetes. As I already have Zipkin locally, I mapped the local port 9412 to port 9411 on the service named `zipkin` in Kubernetes.

If we access the Zipkin portal at `http://localhost:9412/`, we should be able to see the portal for Zipkin running in the AKS cluster even though there is no data, as we still need to configure tracing; we'll do this in the next step.

## Configuring tracing with Zipkin

As we have installed Zipkin, we must now configure Dapr to send all distributed traces to it.

The `\Deploy\configuration-zipkin.yaml` file has a Dapr configuration named `tracing` suitable for our needs, as illustrated in the following code snippet:

```
apiVersion: daprio/v1alpha1
kind: Configuration
metadata:
  name: tracing
  namespace: default
spec:
  tracing:
    samplingRate: "1"
    zipkin:
      endpointAddress: "http://zipkin.default.svc."
```

```
cluster.local:9411/api/v2/spans"

mtls:
  enabled: true
  workloadCertTTL: 24h
  allowedClockSkew: 15m
```

In the previous snippet, `samplingRate` in the tracing element is specified; as the value is `> 0`, Dapr tracing is enabled, and with value `= 1`, all traces get sampled. You can check the Dapr documentation at <https://docs.dapr.io/operations/configuration/configuration-overview/#tracing> to learn more.

Furthermore, in `zipkin`, we configure `endpointAddress` to point to the Zipkin internal service we previously deployed in our Kubernetes cluster.

We apply the configuration with the following command:

```
PS C:\Repos\practical-dapr\chapter11> kubectl apply -f .\
Deploy\configuration-zipkin.yaml
```

We defined a Dapr configuration with Zipkin as the tracing destination. Next, we will enable this configuration in all of our Dapr applications.

## Enabling tracing in Dapr

At this stage, Zipkin is working in Kubernetes, and Dapr is configured to export distributed traces to it. As the last step, we need to start the flow of traces from the Dapr applications.

We now need to update our Dapr applications with the new configuration named `tracing`, as we do in the following example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reservation-service
  namespace: default
  labels:
    app: reservation-service
spec:
  ... omitted ...
  template:
    metadata:
```

```
      labels:
        app: reservation-service
      annotations:
        dapr.io/enabled: "true"
        dapr.io/app-id: "reservation-service"
        dapr.io/app-port: "80"
        dapr.io/config: "tracing"
... omitted ...
```

The change to the `dapr.io/config` annotation must be replicated on each of our Dapr applications' `.yaml` files and then applied, as follows:

```
kubectl apply -f .\Deploy\sample.microservice.order.yaml
kubectl apply -f .\Deploy\sample.microservice.reservation.yaml
kubectl apply -f .\Deploy\sample.microservice.reservationactor.
yaml
kubectl apply -f .\Deploy\sample.microservice.customization.
yaml
kubectl apply -f .\Deploy\sample.microservice.shipping.yaml
```

With the previous `kubectl` commands, we re-apply the deployments of our five Dapr applications in Kubernetes, updated to use the new Dapr configuration.

In *Chapter 10, Exposing Dapr Applications*, we learned how to enable Dapr on an **NGINX** ingress controller and how to integrate Dapr with **Azure API Management** in a self-hosted deployment. Depending on which of these options (or both) you implemented in your environment, the overall solution running in AKS could have one (or two) more Dapr applications on which we should enable the new configuration.

By following the instructions at <https://docs.microsoft.com/en-us/azure/aks/kubernetes-portal?tabs=azure-cli#edit-yaml>, you can edit the deployment manifest to enable the proper configuration with `dapr.io/config` annotation. Do not worry if you do not succeed as it will not dramatically change the way you look at Zipkin traces.

In this chapter, all requests and subsequent Zipkin traces start from the `nginx-ingress` application, representing the Dapr-enabled **NGINX** ingress controller.

### Docker Hub Ready-to-Use Images

As described in *Chapter 9, Deploying to Kubernetes*, ready-to-use container images for each of the sample applications are available on Docker Hub, and deployment scripts are available in the chapter working area.

Deployment script and configuration files are available to support both options - using your own container images from Azure Container Registry or using the ready-to-use container images. For the latter, files are named with the `.dockerhub.yaml` or `.dockerhub.ps1` suffix.

The Dapr dashboard offers a nice view of the configuration, as shown in the following screenshot:

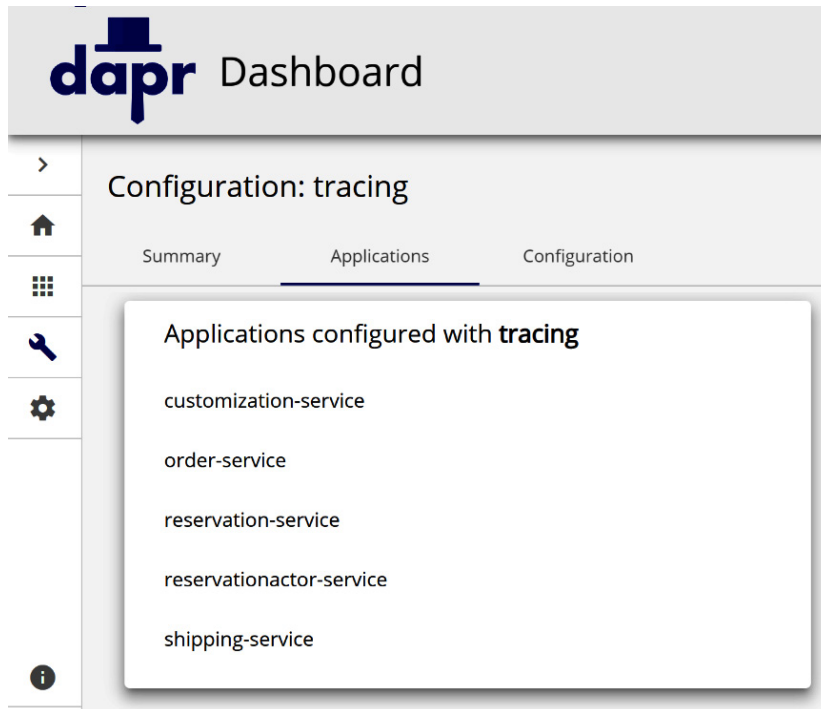


Figure 11.2 – Dapr application adopting tracing

In *Figure 11.2*, we see a configuration named `tracing` and the applications configured with it; each of the Dapr applications in the scope of our solution have tracing enabled.

It's time to learn how Zipkin can help us in understanding our applications' behavior.

## Investigating with Zipkin

We need some data to visualize in Zipkin. The `test.http` file you can use in **VS Code** with the **RestClient** extension is a simple approach to perform sample `http` requests.

In *Chapter 10, Exposing Dapr Applications*, we implemented both an NGINX-based ingress and Azure API Management self-hosted gateway; there should be plenty of options for externally reachable routes to choose from.

Considering the available routes in your specific scenario, by requesting `GET http://<your AKS cluster>/balance/<sku>` and `POST http://<your AKS cluster>/order` a few times, we should generate enough traces to examine in Zipkin.

Let's open the Zipkin portal at `http://localhost:9412/`, or whichever local port we used in the previous section with port forwarding. Let's take a look at the following screenshot:

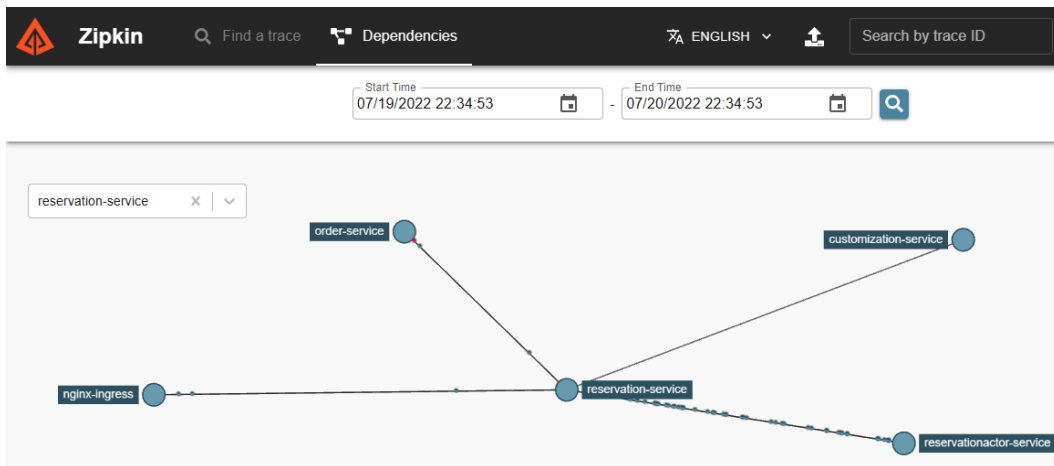


Figure 11.3 – Zipkin dependencies

In *Figure 11.3*, let's examine the dependencies by choosing the service named `reservation-service`, which takes a central position in the flow of interactions in our solution; we should have a similar view with many points (each symbolizing an interaction) from `order-service` to `reservation-service`, many more from here to `reservationactor-service`, and fewer to `customization-service`. Many of the interactions between `reservationactor-service` and `customization-service` are represented with a red color, so it is reasonable to ask: what is going on?

If we used the `test.http` file to simulate requests to our Dapr solution, we would have been ordering and requesting the customization of the infamous `crazycookie` cookie with the **stock-keeping unit (SKU)**; in *Chapter 6, Publish and Subscribe*, we introduced this fixed rule to simulate an unrecoverable error during the customization phase of the **saga pattern** we implemented.

Before we switch to trace analysis in Zipkin, let's clarify two concepts: a **span** is the unit of work executed in a component or service, while a **trace** is a collection of spans. Take a look at the following screenshot:

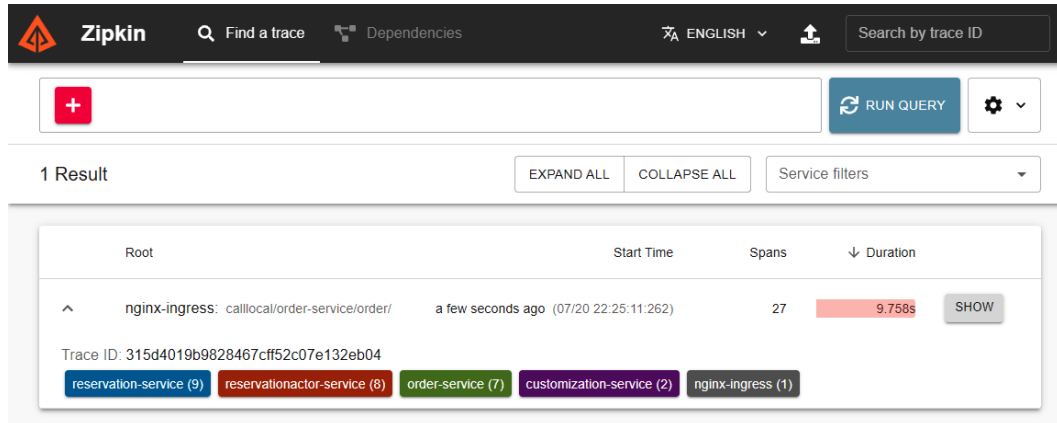


Figure 11.4 – Zipkin traces

In *Figure 11.4*, I managed to find a trace, originating from `nginx-ingress`, as it is the Daprized ingress controller in our solution; it then reaches the solution at `order-service`. If I expand it, I see it includes interactions with all the other Dapr applications; it seems a good candidate for further investigation. By clicking **SHOW**, the Zipkin portal switches to a detailed view of all the spans in the trace.

We should get a view like the one in the following screenshot:

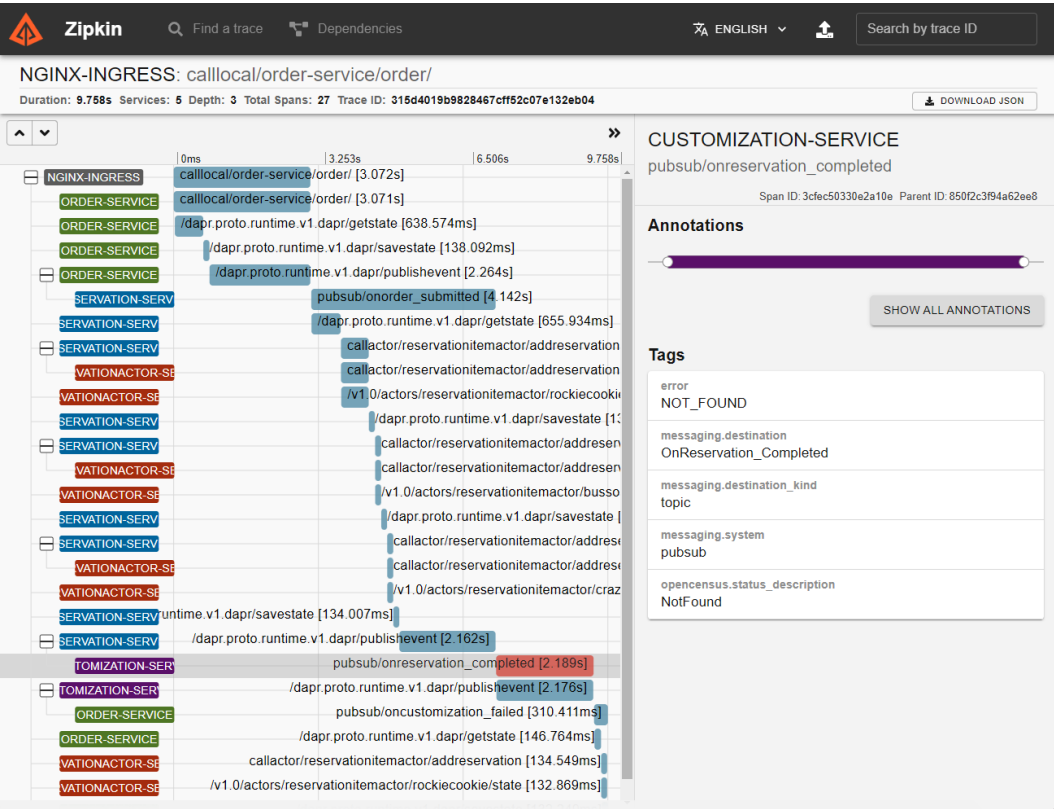


Figure 11.5 – Zipkin trace: NOT\_FOUND error in a span

In *Figure 11.5*, we see the spans displayed in order, showing the service originating the span, the operation, and its execution time.

Let’s focus on a span from `customization-service`, which is represented in red to indicate failure.

In this span, Dapr reports that an error of type `NOT_FOUND` has been received while processing a call to `pubsub/OnReservation_Completed`.

To understand the interaction between Dapr and our code, we need to look at the ASP.NET controller in the project for the customization-service Dapr application—specifically, the `\sample.microservice.customization\Controllers\CustomizationController.cs` file, which you will find in the `chapter09` folder.

Examining the code, we should find the portion in which customization-service simulates an error once customization is requested on a cookie with the `crazycookie` SKU. If the customization fails, our code does the following:

```
... omitted ...
if (!customizedItem.Succeeded)
{
    await daprClient.PublishEventAsync<OrderCustomization>
      (PubSub,common.Topics.
CustomizationFailedTopicName, result);

    Console.WriteLine($"Customization in {order.Id} of {SKU}
      for {quantity} failed");
    return this.NotFound();
}
... omitted ...
```

In the previous code snippet, the ASP.NET controller code is returning a `NotFound` result to the caller, just after it publishes a message to the `OnCustomization_Failed` topic via the configured Dapr pub/sub component — in our case, it is the **Azure Service Bus (ASB)** component.

Our choice in handling the irreversible error that `customer-service` encountered (or simulated) explains the `NotFound` error in the span we see in Zipkin. It also explains the next span we see in order, which is the notification of a failure to the next stages of the saga. Take a look at the following screenshot:

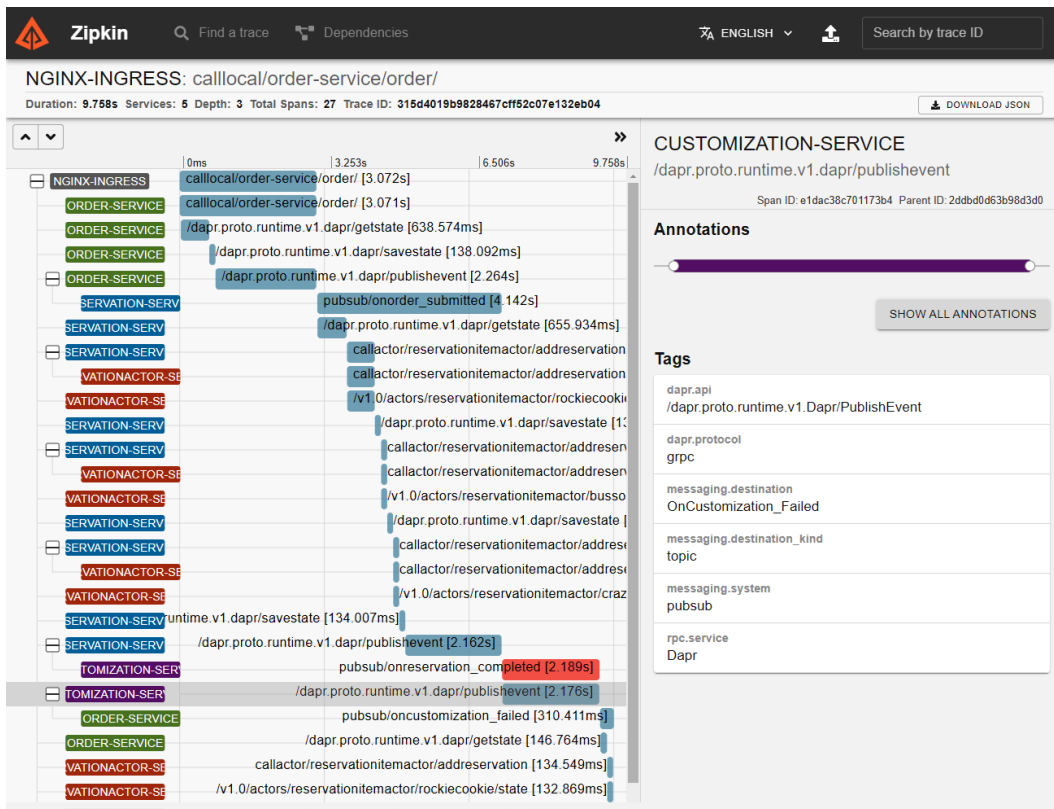


Figure 11.6 – Zipkin trace: span showing a publishevent action

Just to refresh on some of the concepts we learned so far, in *Figure 11.6* we see a span with a call to the Dapr **application programming interface (API)**, using the **gRPC Remote Procedure Call (gRPC)** protocol: it is the Dapr **software development kit (SDK)** for ASP.NET, which relies on gRPC to interact with the Dapr sidecar container.

We have now learned how to enable distributed tracing in our Dapr applications and how to configure Zipkin to help us understand how our application is handling requests and events in complex interactions, as we saw with the saga example.

In the next part of the chapter, we will learn how to monitor all the elements of Dapr, and how they consume resources in a Kubernetes cluster, with Prometheus and Grafana.

## Analyzing metrics with Prometheus and Grafana

**Prometheus** is an open source system and monitoring toolkit, a project with a long history that started in 2012 and is now part of the CNCF.

In our scenario, we will use Prometheus to scrape the metrics exposed by all Dapr Pods and store them as time series. This will act as the data source for the Grafana dashboards.

**Grafana** is an open source visualization and analytics tool. We will use it to examine the Dapr metrics by importing the dashboard templates released by the Dapr project as assets from <https://github.com/dapr/dapr/releases/>.

These are the steps we will follow:

1. Installing Prometheus
2. Installing Grafana
3. Importing dashboards

Let's start by installing the Prometheus service components.

### Installing Prometheus

As described in the Dapr documentation, available at <https://docs.dapr.io/operations/monitoring/metrics/prometheus/>, we should first create a namespace to be used by Prometheus and Grafana, as follows:

```
kubectl create namespace dapr-monitoring
```

We are going to use Helm to install the charts for Prometheus and Grafana in the dapr-monitoring namespace, as illustrated in the following code snippet:

```
helm repo add prometheus-community https://prometheus-  
community.github.io/helm-charts  
helm repo update  
helm install dapr-prom prometheus-community/prometheus -n dapr-  
monitoring
```

The deployment could take some time, so let's proceed to the next command after all the Pods are ready, as follows:

```
kubectl get pods -n dapr-monitoring -w  
NAME                                READY   STATUS  
dapr-prom-kube-state-metrics-7b5b859f9b-
```

```
sjn5x      1/1      Running
dapr-prom-prometheus-alertmanager-676c85b59-
58n77     2/2      Running
dapr-prom-prometheus-node-exporter-
6tt72      1/1      Running
dapr-prom-prometheus-node-exporter-
9n8xf      1/1      Running
dapr-prom-prometheus-node-exporter-
k6bpm      1/1      Running
dapr-prom-prometheus-pushgateway-d5d9dbb
fc-7cpj6   1/1      Running
dapr-prom-prometheus-server-57fbc9446-
8r6rv      2/2      Running
```

The Prometheus Pods are now running; before we move on to Grafana, let's collect the information of the Prometheus service address, as indicated by running the following command:

```
kubectl get svc -n dapr-monitoring
NAME                                TYPE
CLUSTER-IP      dapr-prom-kube-state-metrics
ClusterIP       10.0.176.113   dapr-prom-prometheus-alertmanager
ClusterIP       10.0.122.126   dapr-prom-prometheus-node-
exporter        ClusterIP      None           dapr-prom-
prometheus-pushgateway      ClusterIP      10.0.219.150
dapr-prom-prometheus-server  ClusterIP      10.0.222.218
```

The service is named `dapr-prom-prometheus-server`; we will use this information to configure the Grafana data source in the following section.

## Installing Grafana

Following the Dapr documentation at <https://docs.dapr.io/operations/monitoring/metrics/grafana/>, this is the command to install Grafana:

```
helm repo add grafana https://grafana.github.io/helm-charts
helm repo update
helm install grafana grafana/grafana -n dapr-monitoring
```

As we are using PowerShell, we can obtain the password, autogenerated for us by Grafana, with the following command:

```
$base64secret = kubectl get secret --namespace dapr-monitoring grafana -o jsonpath="{.data.admin-password}"
$password = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($base64secret))
$password
MIDoEFh8YtnfQLByAvG4vB1N4A...
```

The last value from the previous output is the password—in my case, to access the Grafana dashboard. We can now access the Grafana dashboard by port - forwarding to the corresponding Kubernetes service.

## Importing dashboards

We can access Grafana from our local development environment with the following command, mapping local port 8090 (as port 8080 is used by the Dapr dashboard by default) to the remote port 80, like this:

```
kubectl port-forward svc/grafana 8090:80 -n dapr-monitoring
```

By accessing `http://localhost:8090/` and submitting the credentials (the password from the previous section and the default username `admin`), we can log in to Grafana.

Once we access the Grafana portal **user interface (UI)**, we need to add the Prometheus data source first, selecting the **Configuration** dial from the sidebar and **Data Sources**, as illustrated in the following screenshot:

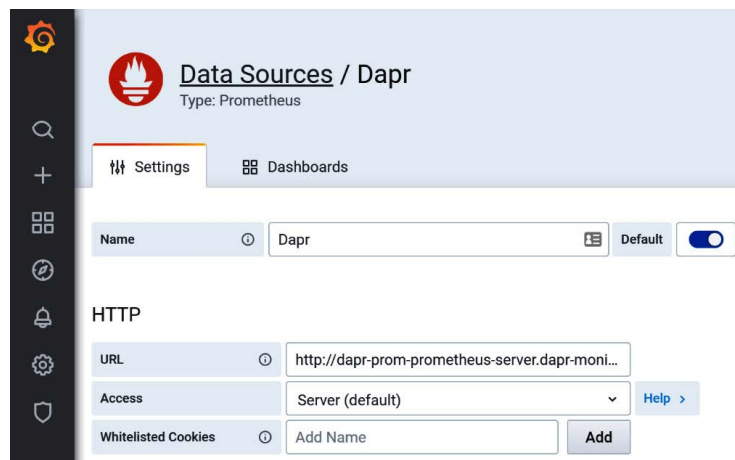


Figure 11.7 – Dapr data source in Grafana

As in Figure 11.7, we added the `http://dapr-prom-prometheus-server.dapr-monitoring` Prometheus service as a data source in Grafana, making it the default one.

We can import the following three ready-to-use dashboards provided by Dapr:

- `grafana-system-services-dashboard.json`
- `grafana-sidecar-dashboard.json`
- `grafana-actor-dashboard.json`

Each Dapr release (<https://github.com/dapr/dapr/releases>) has these dashboards as version-specific assets, and we can find these at <https://github.com/dapr/dapr/tree/master/grafana>.

We can import each one by adding it from the **Create** dial from the sidebar, selecting **Import**, and using the link in the **URL** setting to dashboard it as the source.

The following screenshot shows the **Dapr System Services Dashboard** screen in Grafana:

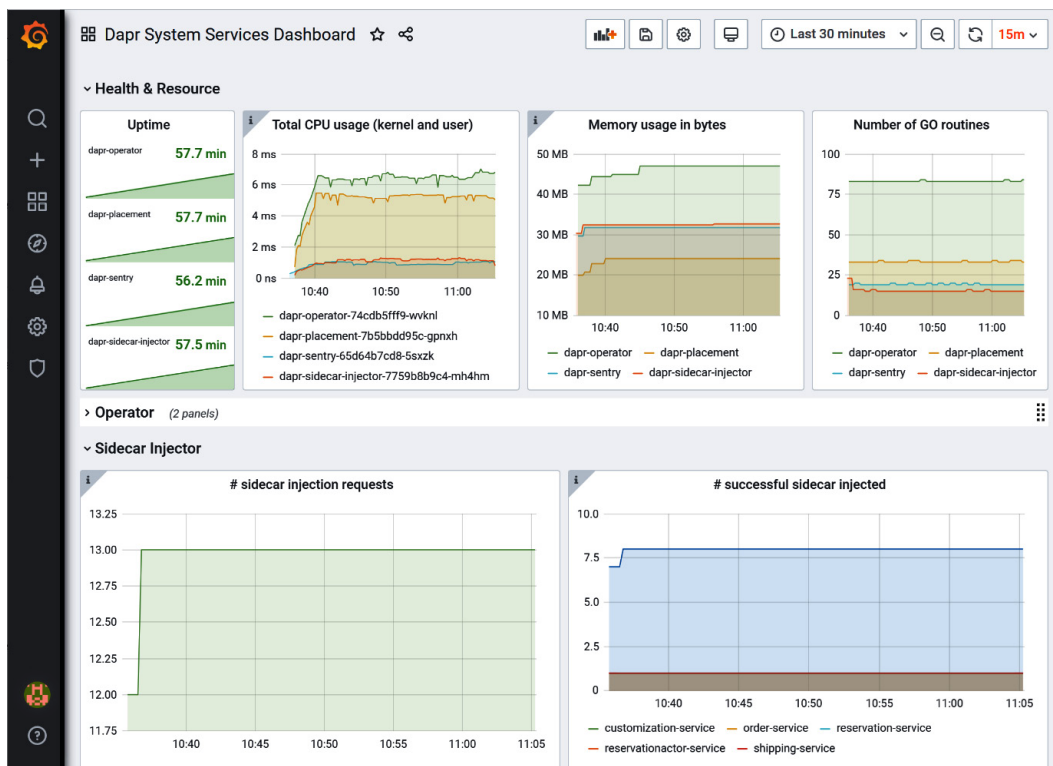


Figure 11.8 – Dapr System Services Dashboard in Grafana

---

At this stage, with Prometheus and Grafana running, we can start exploring the metrics exposed by Dapr. In *Figure 11.8*, we see the system services, with the total CPU and memory used by all Dapr services spread over the various nodes. Similar views are offered for actors and sidecars.

As our solution is not used by external users yet, there is not much activity, other than the one we can simulate by ourselves by sending a few requests to the API. In the next chapter, *Chapter 12, Load Testing and Scaling Dapr*, we will simulate more activity with load-testing tools.

With Prometheus and Grafana, we gained full visibility on how Dapr, from the runtime running in the sidecar of our applications' Pods to the system services, is behaving on Kubernetes.

## Summary

In this chapter, we learned how observability, as provided by Dapr, brings order to the chaotic way a modern cloud-native application could look, if approached with classic tools.

By understanding how Zipkin can help us analyze how our Dapr applications behave in a complex environment such as Kubernetes, we now have the confidence we need to face the brave new world of cloud-native applications.

With Prometheus and Grafana, we learned how Dapr informs developers and operators on how an application is performing on Kubernetes, whether this is a cloud implementation, on-premises, or on the edge.

In the next chapter, we will leverage these abilities to observe how Dapr and our applications react to a heavy user load.



# Load Testing and Scaling Dapr

In this chapter, we will learn how to scale Dapr applications in a Kubernetes environment. After that, we will learn how to load-test a Dapr solution by simulating user behavior with the Locust testing tool.

In this chapter, we will cover the following topics:

- Scaling Dapr in Kubernetes
- Load testing with Locust
- Load testing Dapr
- Autoscaling with KEDA

Load testing is an important practice in software development. It offers developers and operators a scientific approach, guided by practices and tools, to finding the best possible answer to various questions, such as how this application will react to an increase in requests. At which point will the application's response start degrading in terms of success rate and response time? Will the infrastructure be able to sustain a specific rate of requests with a certain level of performance with the allocated resources?

These questions explore both the technical and economic sides of our architecture; in a cloud-native architecture, the operational cost is a factor that can and should influence the design.

## Technical requirements

The code for this chapter's examples can be found in this book's GitHub repository at <https://github.com/PacktPublishing/Practical-Microservices-with-Dapr-and-.NET-Second-Edition/tree/main/chapter12>.

In this chapter, the working area for scripts and code is `<repository path>\chapter12\`. In my local environment, it is `C:\Repos\dapr-samples\chapter12`.

Please refer to the *Setting up Dapr* section in *Chapter 1, Introducing Dapr*, for a complete guide to the tools needed to develop with Dapr and work with the samples.

This chapter builds upon the Kubernetes cluster we set up and configured in *Chapter 9, Deploying to Kubernetes*; refer to this chapter to reach the same configuration.

There are some additional requirements you will need to accomplish the goals of this chapter. Let's take a look.

## Bash

In this chapter, we are going to deploy Azure resources using a shell script.

One option is to install and use **Windows Subsystem for Linux (WSL2)** on Windows 10, by following the instructions at <https://docs.microsoft.com/en-us/windows/wsl/install-win10>.

An alternative to installing **WSL2** locally is to rely on Azure Cloud Shell, as described in <https://docs.microsoft.com/en-us/azure/cloud-shell/quickstart>, and launch the deployment from the context of Azure Cloud Shell.

## Python

Python version 3.7 or later is required if you intend to install and try out Locust on your local development machine.

If this version is not already installed in your environment, you can follow the instructions for Python 3 on Windows at <https://docs.python-guide.org/starting/installation/>.

A quick way to verify whether you have Python 3 installed is to input the following in a Windows terminal:

```
PS C:\Repos\dapr-samples\chapter12> python --version
Python 3.9.0
```

If you decide to use Python from **WSL2**, you should follow the instructions for Python 3 on Linux at <https://docs.python-guide.org/starting/installation/>. A quick way to verify whether you have Python 3 installed is to input the following in a Windows terminal:

```
master@XYZ:/c/Repos/dapr-samples/chapter12$ python3 --version
Python 3.9.0
```

As shown in the preceding output, the version of Python 3 that's been installed on your version of Windows could be different than the one that's available in WSL2.

## Locust

I suggest that you install Locust on your local development machine to verify the tests before publishing them to a Locust swarm.

The installation guide for Locust is available at <https://docs.locust.io/en/stable/installation.html>.

If you decided to use Python from Windows, use the following command to install Locust from a Windows terminal:

```
PS C:\Repos\dapr-samples\chapter12> pip3 install locust
```

To verify the installation and version of Locust, you can use the following command:

```
PS C:\Repos\dapr-samples\chapter12> locust -V
locust 2.10.1
```

If you decide to use Python from **WSL2**, the same context applies to installing Locust:

```
master@XYZ:/c/Repos/dapr-samples/chapter11$ pip3 install locust
```

Locust should be installed in the same environment. This is how you verify the Locust version in WSL2:

```
master@XYZ:/c/Repos/dapr-samples/chapter12$ locust -V
locust 2.10.1
```

Next, let's have a look at the Kubernetes configuration.

## Kubernetes configuration

This chapter builds upon the Kubernetes cluster we set up and configured in *Chapter 9, Deploying to Kubernetes*, and the ingress routes established in *Chapter 10, Exposing Dapr Applications*; please refer to this chapter to ensure you have the same configuration.

The monitoring configuration we prepared with **Zipkin**, **Prometheus**, and **Grafana** in *Chapter 11, Tracing Dapr Applications*, is also useful, even if it's not necessary to follow the instructions. Please refer to this chapter if you want to benefit from the same configuration.

## Scaling Dapr on Kubernetes

In the world of monolithic architectures, the compute and memory resources that are available to an application are constrained by the hosts that it operates on — that is, VMs or physical nodes. For such applications, it becomes an extraordinary challenge to distribute requests and jobs evenly between multiple hosts. They often resort to an active/passive mode in which only a portion of the

allocated resources benefit the application, while the rest are passively sitting idle, waiting for the active environment to fail so that they can switch from their passive role to an active one.

The following diagram depicts the challenges of scaling monolithic applications:

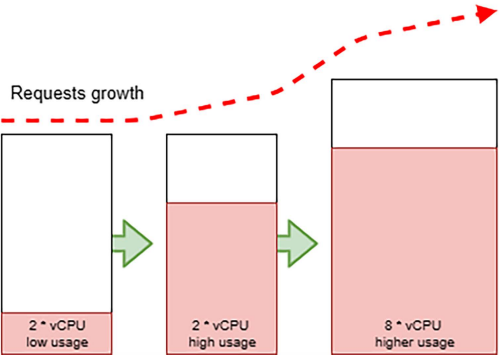


Figure 12.1 – Scaling monolithic applications

Given these factors, to respond to an increase in client requests, which translates to a demand for more computing resources, the response is often to scale up the resources. This can be done by substituting the hosts with more powerful ones, as shown in the preceding diagram. This approach is deemed to inefficiently support workloads with an elastic demand for resources over time.

In the context of microservice architectures, the application is designed with many more components that can be independently deployed and scaled. At any time, there could be multiple instances of a microservice running on different hosts.

For an application based on microservices, **scaling** means reacting to an increase or decrease in resource demand by adding or removing instances of the involved microservices. There is an expectation that the underlying hosting platform offers an elastic pool of resources, as depicted in the following diagram:

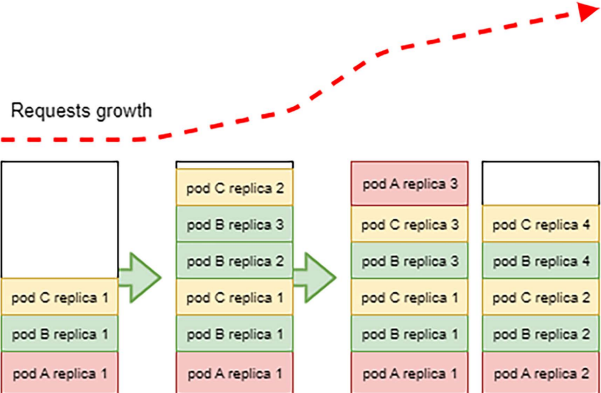


Figure 12.2 – Scaling microservice applications on Kubernetes

As we can see, our Dapr applications are executed as **Pods**, the smallest deployable unit of computing in Kubernetes. We will rely on the features of this powerful orchestrator to scale our microservices.

Moving forward, the following are the concepts we can use to control scaling in Kubernetes:

- Replicas
- Autoscale
- Resource requests and limits

Let's start by exploring the concept of replicas in Kubernetes.

## Replicas

In the context of Kubernetes, scaling translates to increasing the number of Pod instances. Each Dapr application (or any application meant to be deployed on Kubernetes) is configured as a **Deployment** with several **replicas**: this influences the number of **Pods** that get created, each with an instance of a **container** of the application and, in terms of Dapr, a **sidecar**.

One of the benefits of Kubernetes is that you can quickly recover in case of application errors or availability issues with the hosts. We would want to keep our application responsive, even while its components are being recovered. So, in a production-ready environment, we will have more than one replica of each of our microservices; if one fails, the other replica will continue to respond to requests.

The Deployments we've used so far have a `replica` value of 1; only one Pod per Dapr application has been created. This simplifies the initial setup and configuration of any application on Kubernetes, but it is unlikely to stay at 1 once the solution is ready to enter production.

The following is an extract from the `\Deploy\sample.microservice.order.dockerhub.yaml` development file for the `order-service` application:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
  namespace: default
  labels:
    app: order-service
spec:
  replicas: 1
...omitted...
```

Notice that the `replica` value is 1. This might seem contradictory, but let's leave this value as we found it; we will affect it by enabling the autoscale mechanisms offered by Kubernetes.

## Autoscale

In Kubernetes, we can manually change the replicas of a Deployment by updating its configuration or by changing the number of nodes that support the cluster.

Kubernetes can automatically scale up and down the number of replicas of a Deployment, based on the resource usage metrics of its Pods, with the **Horizontal Pod Autoscaler (HPA)**.

Kubernetes can also add or remove cluster nodes, depending on the overall resource's capacity and demand. If there are more Pods to spin up than the available CPUs, the **Cluster Autoscaler (CA)** can sense it and accommodate the request by adding a new node.

The two Kubernetes autoscalers often operate in tandem: the first by requesting more replicas for a Deployment under increasing load, and the second by adapting the cluster size to the overall resource demands by Pods.

You can find more information about these Kubernetes features at <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>, and from an **Azure Kubernetes Service (AKS)** perspective at <https://docs.microsoft.com/en-us/azure/aks/concepts-scale>.

To avoid introducing too much complexity all at once, we will focus on the HPA. There is a valid walk-through available at <https://docs.microsoft.com/en-us/azure/aks/tutorial-kubernetes-scale#autoscale-pods>.

The following are the changes we intend to apply to each of our Dapr application Deployments, starting with `\Deploy\hpa.sample.microservice.order.dockerhub.yaml` for `order-service`:

```
... omitted ...
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: order-service-hpa
  namespace: default
spec:
  maxReplicas: 4
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: order-service
    targetCPUUtilizationPercentage: 50
```

In the previous configuration snippet, we configured a `HorizontalPodAutoscaler` resource so that it scales the replica of Deployment, named `order-service`, from a `minReplicas` value of 1 to a `maxReplicas` value of 4, one replica at a time, if the deployment's resource usage exceeds the `targetCPUUtilizationPercentage` metric of 50%.

However, we are missing an important configuration element of our Dapr application's Deployment: we must specify the resource requests and limits of our Pods.

## Resource requests and limits

Specifying requests and limits for CPU and memory usage is a good practice to allow for proper resource planning in Kubernetes. This prevents your Pods from consuming all the resources that are available on the nodes, which subsequently impacts other Pods in your solution or other systems running in the same cluster. However, this results in instability issues being created for your available nodes and workloads.

By setting resource requests and limits, we inform the Kubernetes cluster of how to properly handle our workload and the basis on which to scale it.

The Pods of our Dapr application have two containers: the Dapr sidecar and our service code, which is based on the ASP.NET container image.

For the CPU and memory requests and limits of our ASP.NET service code, we need to make some initial assumptions that are suitable for a sample we must assess.

The Dapr documentation at <https://docs.dapr.io/operations/hosting/kubernetes/kubernetes-production/#sidecar-resource-requirements> suggests having a configuration that's suitable for the Dapr sidecar container in a production environment.

These settings can be applied to a Dapr application Deployment as annotations. The following `\Deploy\sample.microservice.order.dockerhub.yaml` file for `order-service` shows the intended changes:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
  namespace: default
  labels:
    app: order-service
spec:
  replicas: 1
  selector:
```

```
    matchLabels:
      app: order-service
  template:
    metadata:
      labels:
        app: order-service
      annotations:
        dapr.io/enabled: "true"
        dapr.io/app-id: "order-service"
        dapr.io/app-port: "80"
        dapr.io/config: "tracing"
        dapr.io/log-level: "info"
        dapr.io/sidecar-memory-request: "250Mi"
        dapr.io/sidecar-cpu-request: "100m"
        dapr.io/sidecar-memory-limit: "4000Mi"
        dapr.io/sidecar-cpu-limit: "1"
        ... omitted ...
    spec:
      containers:
        - name: order-service
          image: davidebedin/sample.microservice.order:2.0
          ports:
            - containerPort: 80
          imagePullPolicy: Always
          resources:
            limits:
              memory: "800Mi"
              cpu: "400m"
            requests:
              memory: "200Mi"
              cpu: "100m"
      ... omitted ...
```

With `dapr.io/sidecar-memory-request` and `dapr.io/sidecar-cpu-request`, we are specifying that the Dapr sidecar in a Pod for `order-service` should start by requesting 250 MiB of memory and 100 m of CPU (0.1 vCPU).

The container that contains our service code (which is using the `<your ACR registry name>.azurecr.io/sample.microservice.order:2.0` image or `davidebedin/sample.microservice.order:2.0`) is requesting 100 MiB of memory and 100 m of CPU.

### Docker Hub Ready-to-Use Images

As described in *Chapter 9, Deploying to Kubernetes*, ready-to-use container images for each of the sample applications are available on Docker Hub, and their deployment scripts are available in this chapter's working area.

Deployment scripts and configuration files are available to support both options — using your own container images from Azure Container Registry or using the ready-to-use container images. For the latter, files are named with the `.dockerhub.yaml` or `.dockerhub.ps1` suffixes.

In this chapter, it is assumed you will prefer to leverage the ready-to-use images in Docker Hub.

The Dapr sidecar has much higher resource limits than our service code since it's going to handle most of the I/O for our application.

See <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#resource-units-in-kubernetes> for more information on the units of measure that are used in Kubernetes.

Once we are done, we can apply the configuration, as described in the `Deploy\deploy-solution.dockerhub.ps1` file:

```
kubectl apply -f .\Deploy\sample.microservice.order.dockerhub.yaml
kubectl apply -f .\Deploy\sample.microservice.reservation.dockerhub.yaml
kubectl apply -f .\Deploy\sample.microservice.reservationactor.dockerhub.yaml
kubectl apply -f .\Deploy\sample.microservice.customization.dockerhub.yaml
kubectl apply -f .\Deploy\sample.microservice.shipping.dockerhub.yaml
```

With the preceding commands, we applied the aforementioned changes to all our application deployments.

With the following command, we can verify that the HPAs for our Dapr application have been configured:

```
C:\Repos\practical-dapr\chapter12> kubectl get hpa
```

NAME	REFERENCE
customization-service-hpa	Deployment/customization-service
13%/50%	1
4	2m21s
order-service-hpa	Deployment/order-service
15%/50%	1
4	2m28s
reservation-service-hpa	Deployment/reservation-service
14%/50%	1
4	2m25s
reservationactor-service-hpa	Deployment/reservationactor-ser-
vice	
20%/50%	1
4	2m23s
shipping-service-hpa	Deployment/shipping-service
3%/50%	1
4	2m18s

An HPA analyzes the metrics of the resources that are used by the Pods in a Deployment. Once it detects an increase in CPU usage that's beyond the configured threshold, it triggers an increase in replicas.

The load test we will set up in the next configuration will trigger this mechanism.

## Load testing with Locust

Load testing is a practice in software development that's used to determine the performance of a complex system under load. This is generated by simulating the concurrent access of users. Load testing web resources, such as an API, usually requires multiple agents to be orchestrated, each with enough internet bandwidth and compute resources to simulate the activity of many users.

In our scenario, we plan to verify the performance and capabilities of our *Biscotti Brutti Ma Buoni* sample backend, implemented with Dapr and running on Kubernetes.

**Locust** is a popular open source load-testing tool. It allows you to define a user's behavior with simple Python scripts and distribute these on as many **worker nodes** as needed, orchestrated by a **master node**. More information is available at <https://locust.io/>.

### Important note

I learned about Locust from a GitHub repository (<https://github.com/yorek/locust-on-azure>) that was created by my colleague Davide Mauri, PM of the Azure SQL team at Microsoft. This repository offers deployment scripts that allow you to use **Azure Container Instances (ACI)** as a compute option for Locust masters and workers. I personally contributed to the repository with a virtual network - integrated deployment option.

The **Locust on Azure** repository has already been copied to this chapter's base directory. You can clone it by using the following command:

```
PS C:\Repos\dapr-samples\chapter12> git clone https://github.com/yorek/locust-on-azure.git
```

Once the repository has been cloned, copy the Locust test file, prepared for our scenario, over the default test file provided in the repository we just cloned in the previous step:

```
PS C:\Repos\dapr-samples\chapter12> copy .\loadtest\locustfile.py .\locust-on-azure\locust
```

Let's open the `locust-on-azure\locust\locustfile.py` file to examine the Locust test for our scenario:

```
from locust import HttpUser, TaskSet, task, between
... omitted ...
class APIUser(HttpUser):
    wait_time = between(0.1, 1)
    @task(50)
    def getbalance(self):
        SKU = RandomSKU()
        with self.client.get("/balance/%s" % SKU,
                             name="balance", catch_response=True) as response:
            if (not(response.status_code == 201 or 200)):
                response.failure("Error balance: %s" %
                                response.text)
    @task(1)
    def postorder(self):
        http_headers = {'content-type': 'application/json'}
        payload = RandomOrder()

        with self.client.post("/order", json=payload,
                              headers=http_headers, name="order",
                              catch_response=True) as response:
            if (not(response.status_code == 201 or 200)):
                response.failure("Error order: %s" % response.
                                text)
```

From the preceding Python snippet, we can see a class arbitrarily named `APIUser`; Locust will use each class to represent user behavior. In our test, we have only one, which, with a delay between 100 ms and 1 second, as expressed by `wait_time = between(0.1, 1)`, executes the methods decorated with `@task`. There are two methods, each with a different weight: `getbalance`, for reading the balance of a product by its SKU, and `@task(50)`, which has a 50 times higher chance of being executed than `postorder`, a method used to simulate an order submission.

As we saw previously, a Locust test is a simple `.py` Python file. For further information on this, it is worth looking at the quickstart presented at <https://docs.locust.io/en/stable/quickstart.html>.

In *Chapter 10, Exposing Dapr Applications*, we both implemented an NGINX-based ingress and Azure API Management self-hosted gateway; the Locust test presented earlier relies on the GET `http://<your AKS cluster>/balance/<sku>` and POST `http://<your AKS cluster>/order` routes, but you can change the test file to leverage the Azure API Management supported routes.

Now that we've looked at the Locust test definition and have a suitable example ready, we can launch it to assess the performance of our Dapr applications.

## Load testing Dapr

Before we activate a more complex test infrastructure for Locust on ACI, it is best to first check, and eventually debug, the Locust test locally.

The following steps take us through preparing the data to enabling autoscaling on a running load test:

1. Preparing the data via port-forwarding
2. Testing Locust locally
3. Locust on ACI
4. Configuring the HPA

First, we need to make sure the data in our environment can support our scenario. This is what we'll do in the next section.

### Preparing the data via port-forwarding

Before we launch the test, which is readily available at `locust-on-azure\locust\locustfile.py`, let's focus on an aspect we oversaw; the SKU for the cookies is randomly composed from **cookie001** to **cookie999**, with the addition of the infamous **crazycookie**, the main actor (pun intended) of the saga from *Chapter 6, Publish and Subscribe*.

The code that's being used for our sample Dapr applications is extremely permissive; you could order an item with a negative balance without an issue. You can also add both validation and business logic to it if you are interested.

Nevertheless, all the `getbalance` test requests will fail since the items that have been created by these SKUs are unknown by the Dapr applications. We can prepare the data by launching the `C:\Repos\dapr-samples\chapter12\loader\generatedata.py` Python file.

There is nothing special in this data generator script, other than that it relies on reaching `ReservationItemActor`, which is not exposed via an ingress controller, at the standard Dapr API URL of `http://localhost:5001/v1.0/actors/ReservationItemActor/{id}/method/AddReservation`, with `{id}` being the random cookie SKU.

To use the script, we need to use port-forwarding to locally map port 5001 to the Dapr sidecar on port 3500 in any of the Pods containing our Dapr applications in Kubernetes.

With the following command, you can obtain the name of a Pod in Dapr:

```
PS C:\Repos\dapr-samples\chapter12> kubectl get pods -l
app=order-service
```

NAME	READY	STATUS	RESTARTS	AGE
order-service-75d666989c-zpggg	2/2	Running	0	23h

With the previous command, I searched for Pods with a specific label assigned to `order-service` Dapr applications.

```
PS C:\Repos\dapr-samples\chapter12> kubectl port-forward order-
service-75d666989c-zpggg 5001:3500
Forwarding from 127.0.0.1:5001 -> 3500
Forwarding from [::1]:5001 -> 3500
```

In the previous command snippet, with port-forwarding, we can gain access to a Dapr sidecar running inside Kubernetes. We can launch the `C:\Repos\dapr-samples\chapter12\loader\generatedata.py` script from VS Code or a Windows terminal session with the following command:

```
PS C:\Repos\dapr-samples\chapter12\loader> python .\
generatedata.py
```

Once all 1,000 instances of **ReservationItemActor** have been created, we can proceed with the Locust tests.

## Testing Locust locally

To launch the `locust-on-azure\locust\locustfile.py` test file with Locust, use the following command:

```
PS C:\Repos\dapr-samples\chapter12\locust-on-azure\locust>
locust -H http://<ingress>.<aks url>
```

The `-H` parameter sets the host address. Since the tests invoke `/balance` and `/order`, the host should contain the base path, including the domain name that's been defined for the ingress controller.

For our first run, which will tell us whether the tests are correct, let's simulate a few users with a slow spawning rate. If we do not face any unexpected exceptions, meaning that there are no bugs in our test code, we can increase the number of users.

Let's ramp up the test to 1,000 users!

In the following screenshot, you can start to see why executing a load test from a single node is not a good idea:

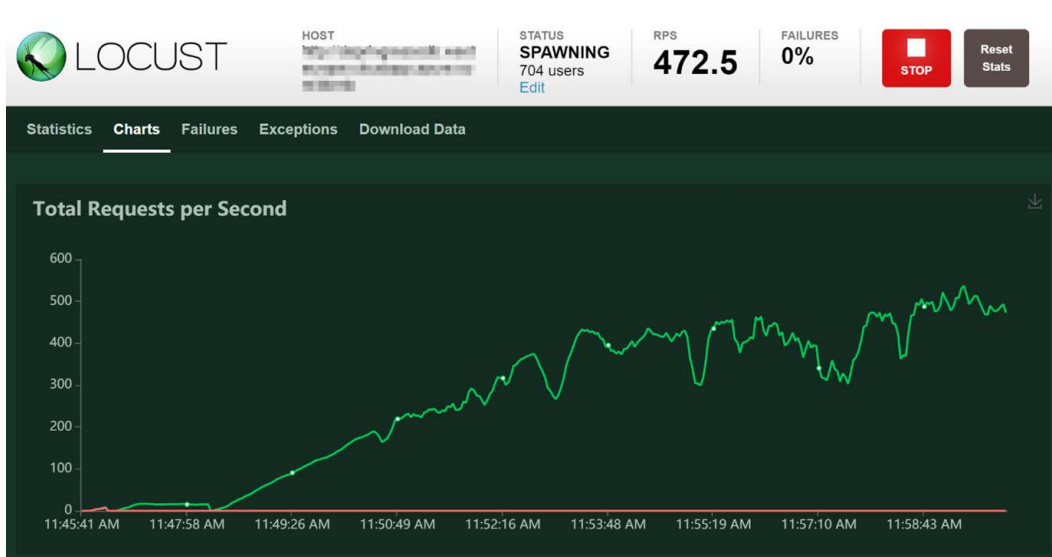


Figure 12.3 – Locust RPS encountering high CPU usage

In the preceding screenshot, you can see the Locust dashboard. It shows how the **requests per second (RPS)** have reached a plateau of **400 RPS** and that they are not moving from there. From a Kubernetes perspective, I did not see any high CPU utilization.

Once I looked at the Windows terminal session running Locust, I saw the following output:

```
PS C:\Repos\dapr-samples\chapter12\locust-on-azure\locust>
locust -H http://<ingress>.<aks url>/bbmb
[2020-10-31 11:46:56,730] XYZ/INFO/locust.main: Starting web
interface at http://0.0.0.0:8089 (accepting connections from
all network interfaces)
[2020-10-31 11:46:56,746] XYZ/INFO/locust.main: Starting Locust
1.3.1
[2020-10-31 11:48:21,076] XYZ/INFO/locust.runners: Spawning
1000 users at the rate 1 users/s (0 users already running)...
[2020-10-31 11:51:57,393] XYZ/WARNING/root: CPU usage above
90%! This may constrain your throughput and may even give
inconsistent response time measurements! See https://docs.
locust.io/en/stable/running-locust-distributed.html for how to
distribute the load over multiple CPU cores or machines
```

I’m running Locust from my local development machine to check the quality of the tests. However, as reported in the output, I already encountered a CPU with high-stress conditions.

Considering the high CPU usage warning, in conjunction with the time series shown in the preceding screenshot, this could well be a factor that’s negatively impacting the validity of the overall test.

Now that we’ve stopped the load test, let’s focus on the metrics presented by Locust:

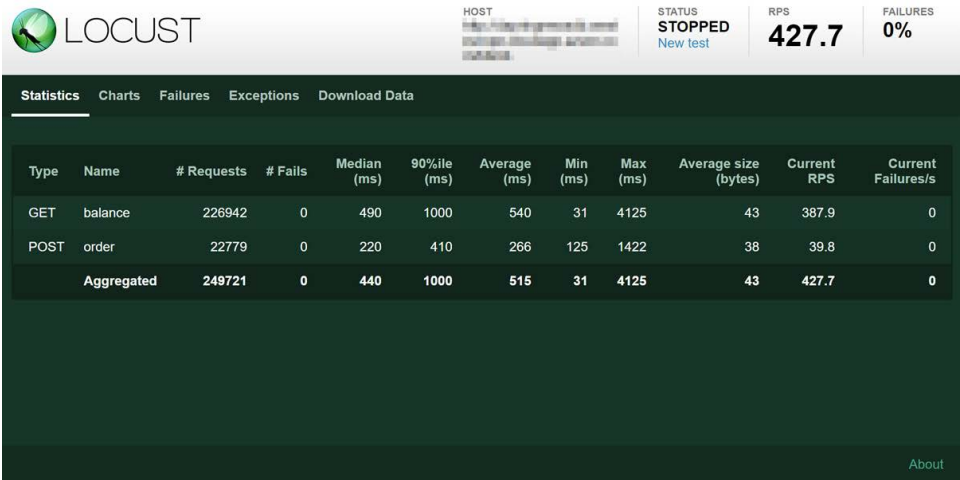


Figure 12.4 – Locust test statistics

In the preceding screenshot, you can appreciate the data presented by Locust on the **Statistics** pane, which is where the performance of each task is presented. This information, including **Exceptions** and **Failures** reports, is also available as CSV files that you can download.

Your experience may change, depending on the CPU and network bandwidth of the local machine you are using for Locust, as well as the network capacity of your site.

In this section, we learned how to write a Locust test, how to test it locally, how to interpret the Locust metrics, and, most importantly, why an Azure-based deployment of a Locust swarm is a good approach so that we have an effective load testing environment.

## Locust on Azure Container Instances

The Locust on Azure implementation offered at <https://github.com/yorek/locust-on-azure> leverages the **ACI** resource to execute Locust in distributed mode, along with master and worker nodes.

While AKS offers containers orchestrated by Kubernetes, ACI allows us to execute individual containers without the need for an orchestrator.

The following diagram shows how this solution is composed:

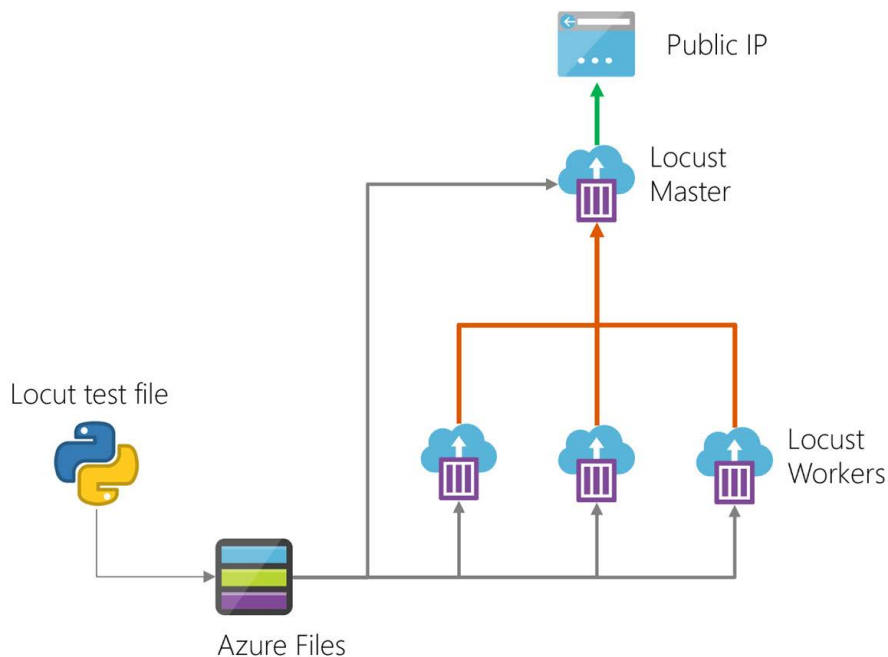


Figure 12.5 – Locust on ACI

The template is activated by executing the `\locust-on-azure\azure-deploy.sh` script, which can be found in this chapter's base directory. The script uploads the Python files present in the `\locust-on-azure\locust` directory to a newly created Azure Files storage. This storage option in Azure is mounted on all ACI instances. Finally, Locust is activated on the ACI, with the worker nodes interacting with the master node. This is the one exposing the Locust portal to the users, as shown in the preceding diagram.

Locust on Azure is a shell script; therefore, we need to shift to **WSL**, as described in the *Technical requirements* section. We also need to access our Azure subscription from this context with `az login`. Once we've done this, we can launch the `azure-deploy.sh` script:

```
master@XYZ:/c/Repos/dapr-samples/chapter12/locust-on-azure$ ./
azure-deploy.sh
Environment file not detected.
Please configure values for your environment in the created
.env file and run the script again.
TEST_CLIENTS: Number of locust client to create
USERS_PER_CLIENT: Number of users that each locust client will
simulate
SPAWN_RATE: How many new users will be created per second per
locust client
HOST: REST Endpoint to test
RESOURCE_GROUP: Resource group where Locust will be deployed
AZURE_STORAGE_ACCOUNT: Storage account name that will be
created to host the locust file
```

As described in the first execution, we need to specify a few variables in order to describe the destination environment. Once we've specified those in the `.env` files in the same `\locust-on-azure\locust` directory, the setup will be complete, and the resources will be successfully deployed to Azure:

```
master@XYZ:/c/Repos/dapr-samples/chapter12/locust-on-azure$ ./
azure-deploy.sh
loading from .env
starting
creating storage account: daprllocustsharedstorage
retrieving storage connection string
creating file share
uploading simulator scripts
uploading /c/Repos/dapr-samples/chapter12/locust-on-azure/
```

```
locust/locustfile.py
Finished[#####] 100.0000%
deploying locust (10 clients)...
locust: endpoint: http://<omitted>:8089
locust: starting ...
locust: users: 1000, spawn rate: 10
```

Once we've deployed the script, we receive the endpoint of our Locust portal.

Now that we have a proper testing platform for a Locust swarm running on Azure, unbridled from the CPU and bandwidth constraints of a single workstation, we are ready to launch a new test.

## Observing the Horizontal Pod Autoscaler

In this chapter, we learned how an HPA operates, and we accordingly configured our Dapr applications with autoscalers. Now, it is time to see it in action.

Let's launch the Locust test from the portal. This time, it can be reached at the public IP of the master ACI, as printed in the output shown in the previous section.

Since we are aiming to apply a load of 1,000 concurrent users to our API, let's examine the behavior of the HPA with the `kubectl get hpa -w` command while the Locust swarm starts executing requests:

```
PS C:\Repos\dapr-samples\chapter12> kubectl get hpa -w
```

NAME	TARGETS	REPLICAS	AGE
customization-service-hpa	78%/50%	1	14m
order-service-hpa	139%/50%	1	14m
reservation-service-hpa	185%/50%	4	14m
reservationactor-service-hpa	147%/50%	1	14m
shipping-service-hpa	2%/50%	1	14m
reservation-service-hpa	255%/50%	1	14m
customization-service-hpa	78%/50%	2	14m
order-service-hpa	139%/50%	3	15m
reservationactor-service-hpa	142%/50%	4	14m
reservation-service-hpa	255%/50%	4	15m
customization-service-hpa	10%/50%	2	15m
order-service-hpa	29%/50%	3	15m
reservationactor-service-hpa	66%/50%	8	15m
shipping-service-hpa	2%/50%	1	15m

reservation-service-hpa	142%/50%	8	15m
reservationactor-service-hpa	66%/50%	10	15m
reservation-service-hpa	142%/50%	10	15m

A few columns have been removed from the preceding output to evidence the growth in replicas enforced by the HPAs. This was triggered by an increase in CPU usage by the `reservation-service` and `reservationactor-service` applications. These Dapr applications perform most of the operations in the overall solution, so this behavior was expected.

Let’s examine the same data, but plotted on a chart:



Figure 12.6 – HPA scaling up and down

As we can see, the HPA for the `reservation-service` and `reservationactor-service` applications scaled up quickly to 10 Pod instances and was kept at the maximum limit imposed by the HPA for the duration of the test. Once they had done this, they scaled down during the cooling period to the minimum number of Pods — that is, 1:

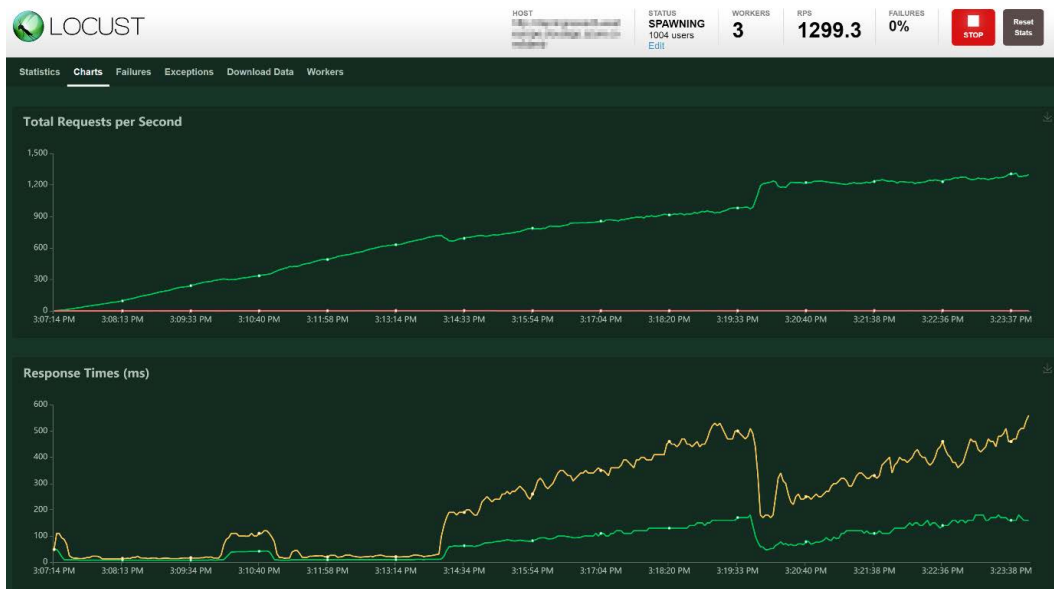


Figure 12.7 – Locust load test

There are several other aspects to account for during a load test: the impact on the Dapr state store and publish/subscribe is extremely relevant. In the preceding screenshot, we can see a few spikes in the response time that should be investigated.

### Resiliency in Dapr

As we explored the resiliency feature throughout the book, we learned how it can be applied to the building blocks used in our solution: service-to-service invocation, state component, publish and subscribe component, and actor.

What happens if a Dapr application scale based on demand publishes messages another Dapr application subscribes to, but the latter application experiences transient errors from the state management because of excessive load on the external database? How do we want to handle it?

Resiliency applies the end-to-end scenario as an issue in one Dapr application, while it could leave other ones apparently unaffected, and might increase the time needed to complete the operation, having a cascading effect on the number of operations completed in a unit of time. A load test with lots of positive results might not tell the whole story; by providing sophisticated policies to influence how Dapr and our application recover from errors, we gain the ability to handle complex scenarios.

By looking at the Locust portal, we can see that setting a higher number of Pods could have had a positive impact.

If our load test was aiming to allow 1,000 concurrent users to interact with our Dapr application's API with an end-to-end response time of about half a second, having nearly 1,200 requests per second could be a satisfactory result.

Before we complete our load testing journey, let's consider another autoscaling option.

## Autoscaling with KEDA

So far, we've learned that the HPA is triggered by the CPU and memory metrics of the Pods in a Deployment.

**Kubernetes-Based Event-Driven Autoscaling (KEDA)** is a **Cloud-Native Computing Foundation (CNCF)** project, with the objective of extending the capabilities of the Kubernetes HPA so that it reacts to the metrics of resources that are external to the Kubernetes cluster.

You can learn more about KEDA (<https://keda.sh/>) in the context of Dapr at <https://docs.dapr.io/developing-applications/integrations/autoscale-keda/>.

Considering the vast adoption of the publish/subscribe Dapr building block in our example, it would be smart to increase (and decrease) the number of Pods based on the messages accumulating in the underlying messaging system, which is Azure Service Bus in our case. If the number of enqueued messages grows, we could add more Pods so that Dapr dequeues the messages and our ASP.NET code processes the requests.

In more general terms, reacting to the metrics of the incoming messages is useful if we wish to anticipate a load other than the stress that occurs in processing.

KEDA offers scalers for most of the publish/subscribe components supported by Dapr, which makes it a powerful tool at our disposal.

We will see KEDA scalers in action in *Chapter 13, Leveraging Serverless Containers with Dapr*.

## Summary

In this chapter, we learned how to scale the Kubernetes resources of our Dapr applications with the Deployment configuration, as well as how to automatically adapt the number of replicas to CPU and memory usage with the HPA.

The concepts we explored in this chapter gave us a more solid approach to testing Dapr applications under specific conditions; is our overall solution, starting with the nodes of the Kubernetes cluster and including the database (state store) and message bus (publish/subscribe), capable of sustaining a specific load?

Even if we had to venture outside the land of C# and .NET to leverage Locust, I think the advantages of learning a popular, developer-oriented load-testing framework justify the effort. Python is also supported in Dapr with an SDK for services and actors, so maybe this could be the next stage of our learning experience with Dapr.

# Leveraging Serverless Containers with Dapr

In this chapter, at the end of our journey of discovery of Dapr, we will learn how to deploy our containerized application to **Azure Container Apps**: a fully managed serverless container service in Microsoft Azure.

With the newly acquired experience of developing microservices applications with Dapr and operating it on Kubernetes, we will be able to appreciate how Azure Container Apps simplifies the developer's job by giving us all the benefits of a container orchestrator without the burden of operating on a Kubernetes cluster.

This chapter will help us understand the business and technical advantages offered by a serverless container service.

In this chapter, we will cover the following topics:

- Learning about the Azure Container Apps architecture
- Setting up Azure Container Apps
- Deploying Dapr with Azure Container Apps
- Autoscaling Azure Container Apps with **Kubernetes Event-Driven Autoscaling (KEDA)**

As developers, we want to focus our attention and effort on the definition and implementation of an application. However, it is equally as important to comprehend how to operate our solution on the infrastructure and services at our disposal.

Our first objective is to understand the architecture of Azure Container Apps and find out how it can benefit the *Biscotti Brutti Ma Buoni* (Italian for “ugly but good cookies”) fictional solution we built so far.

## Technical requirements

The code for the examples in this chapter can be found in this book's GitHub repository at <https://github.com/PacktPublishing/Practical-Microservices-with-Dapr-and-.NET-Second-Edition/tree/main/chapter13>.

In this chapter, the working area for scripts and code is expected to be `<repository path>\chapter13\`. In my local environment, it is `C:\Repos\dapr-samples\chapter13`.

Please refer to the *Setting up Dapr* section in *Chapter 1, Introducing Dapr*, for a complete guide on the tools needed to develop with Dapr and work with the examples in this book.

In this chapter, we will also reuse Locust: to refresh that knowledge, please refer to *Chapter 12, Load Testing and Scaling Dapr*.

## Learning about the Azure Container Apps architecture

Azure Container Apps is a managed serverless container service on the Microsoft Azure cloud.

It is managed in the sense that you, the user, can configure the service to fit your needs but you are not involved in the management of the underlying resources; it is serverless as you do not have to care about the characteristics and scaling of the VMs used by the underlying hosting platform.

With Azure Container Apps, your application does run on an **Azure Kubernetes Service (AKS)** cluster managed on your behalf by the Azure platform: you will not have to configure and maintain the resource by yourself.

Here are the three main components of Azure Container Apps:

- **KEDA** is already configured, ready to be used to scale your workload based on HTTP requests, CPU, and memory usage, or any of the other scale triggers supported by KEDA
- **Dapr** is integrated into Azure Container Apps, so it can be leveraged by any container workload already adopting it
- **Envoy** is an open source proxy designed for cloud-native applications and is integrated into Azure Container Apps to provide ingress and traffic-splitting functionalities

### Cloud-native

As described at <https://www.cncf.io/>, the **Cloud Native Computing Foundation (CNCF)** serves as the vendor-neutral home for many of the fastest-growing open source projects. KEDA, Envoy, and Dapr are all projects under the CNCF umbrella.

Azure Container Apps provides the following functionalities:

- Autoscales based on any KEDA scale trigger
- Provides HTTPS ingress to expose your workload to external or internal clients
- Supports multiple versions of your workload and splits traffic across them
- Observes application logs via Azure Application Insights and Azure Log Analytics

As we can see from the following diagram, the **environment** is the highest-level concept in Azure Container Apps: it does isolate a solution made of multiple container apps from other workloads, from a networking, logging, and management perspective:

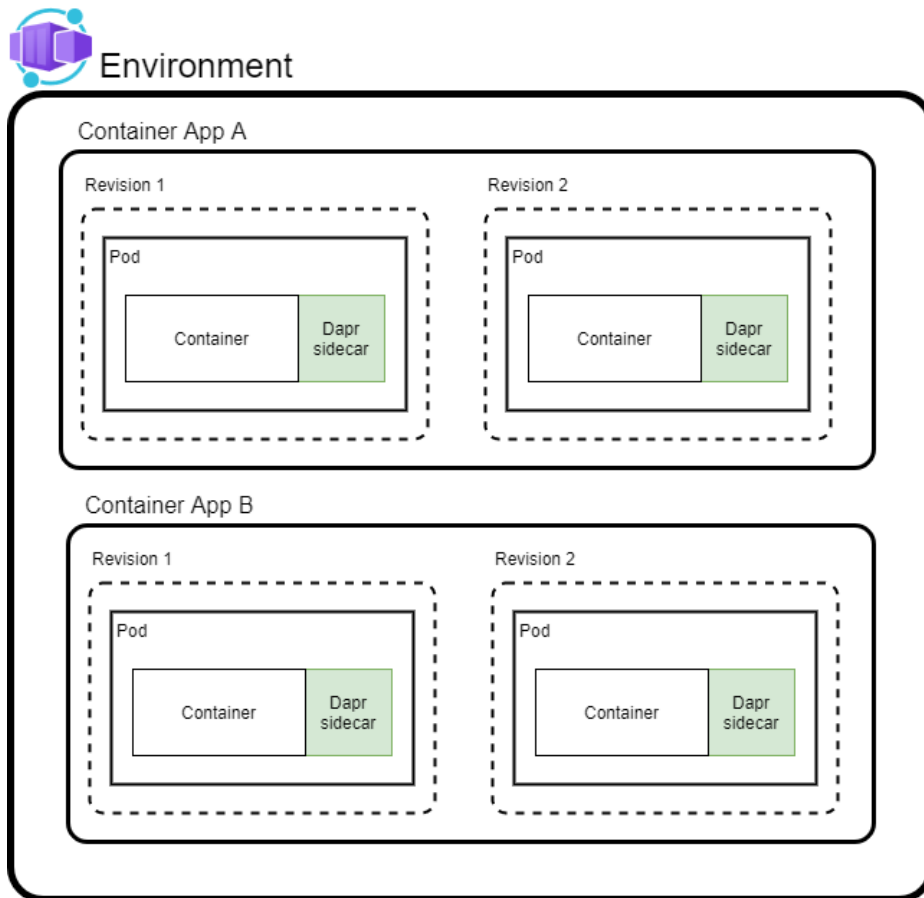


Figure 13.1 – Azure Container Apps architecture

As depicted in *Figure 13.1*, an environment can contain multiple **container apps**: we can already guess each of the microservices in our solution will become a distinct container app.

These are the core concepts in container apps:

- A **revision** is an immutable snapshot of our container app: each container app has at least one, created at first deployment, and you can create new revisions to update a new version of your container app
- A **Pod** is a familiar concept from the Kubernetes architecture: in the context of Azure Container Apps, it does not surface as a keyword in the CLI or portal but, as AKS is supporting our workload under the covers, we know that a container image of a specific revision is being executed in this context
- A **container** image can originate from any container registry: over the course of this book, we used Azure Container Registry but, in this chapter, we will leverage the readily available images in Docker Hub
- As **Dapr** is already configured in the Azure Container Apps service, the sidecar can be enabled in any container app

Before we dive into the configuration details of Azure Container Apps, let us explore why should we take this option into consideration for our solution.

## Understanding the importance of Azure Container Apps

Let us recap the progress made so far with our fictional solution. We managed to create a fully functional solution with Dapr, and we successfully deployed it to AKS. Why should we care about Azure Container Apps, the newest service for containers on the Azure cloud?

Please bear with me as I expand the fictional scenario at the core of the sample in this book. Driven by the success of *Biscotti Brutti Ma Buoni* e-commerce solution we built so far, the capabilities of made-to-order cookie customization gained the attention of other businesses: the company behind *Biscotti Brutti Ma Buoni* decides to market the in-house developed solution to other businesses, selling it as a **Software as a Service (SaaS)** in tandem with the equipment to run a shop. Given this business objective, how can we accomplish it?

Starting with *Chapter 9, Deploying to Kubernetes*, we gained experience with AKS; over *Chapter 10, Exposing Dapr Applications*, we learned a few alternative approaches for exposing our applications in AKS; and in *Chapter 11, Tracing Dapr Applications*, we explored how to configure a monitoring stack in AKS and integrate it with Dapr. Finally, in *Chapter 12, Load Testing and Scaling Dapr*, we explored how to scale our application to respond to increased client load with a simple **Horizontal Pod AutoScaler (HPA)**.

Kubernetes is a powerful container orchestrator, and AKS can significantly simplify its management. Nevertheless we, as developers and architects, must deal with many tools (**kubect**l, **Helm**, **YAML**, **Prometheus**, **Grafana**, and more) and concepts such as *Pod*, *service*, *Deployment*, *replica*, and *ingress controller* that help us get the work done but do not add any value to our solution.

The *Biscotti Brutti Ma Buoni* company adopted AKS and integrated it with the IT infrastructure to support our solution and many other business applications as well. The operations team can maintain the internal and externally facing applications at the **business continuity (BC)** level required by the company.

Offering a solution as SaaS to external customers is a completely different story; the level of service you must guarantee to your customers might require a different effort in development and operations, probably higher than what you can accept for your own internal usage.

Given the experience with AKS, the solution team has the following open questions:

- Do we extend the existing AKS cluster to SaaS customers?
- How challenging would it be to support additional AKS clusters, devoted to SaaS customers?
- How do we manage the multi-tenancy of our solution on Kubernetes?
- How can we combine internal DevOps practices with what is required for SaaS customers?

After a thorough conversation involving all the company stakeholders, the company comes up with the decision to not keep the existing AKS infrastructure separate from the new SaaS initiative. The operational model achieved is fit for the company's internal operations.

The idea of adding AKS clusters to isolate SaaS customers does not attract the favor of the IT and operations team. Furthermore, even if this goal can be achieved, how can they adapt the solution to support multiple tenants? There are approaches to support multi-tenancy in Kubernetes; it is likely this requirement would increase the operational complexity.

Answering these questions triggers new questions. The consensus is that it would be best to keep the solution on the existing evolution path, hopefully finding a way to enable multi-tenancy without adding complexity to the toolchain. Also, no stakeholder finds any value in managing Kubernetes if not strictly necessary: *Biscotti Brutti Ma Buoni* is a purely cloud-native solution, and the company would like to venture into the SaaS market without the risk of turning into a **managed service provider (MSP)**.

With Azure Container Apps, the *Biscotti Brutti Ma Buoni* solution can be deployed as is, with no changes to the code or containers. An isolated environment can be provisioned for each SaaS customer with no additional complexity. The rich scaling mechanisms enable a company to pay for the increase in resources only when the solution needs it. In a nutshell, all the benefits of running containers without any concerns about managing Kubernetes.

## Container options in Azure

In this book, we explored two options to support a container workload: AKS and Azure Container Apps. I highly recommend reading this article to compare Azure Container Apps with the other Azure container options: <https://docs.microsoft.com/en-us/azure/container-apps/compare-options>.

To recap, the reason the team behind *Biscotti Brutti Ma Buoni* decided to adopt Azure Container Apps can be condensed into a single sentence: to achieve the flexibility of microservices running in a Kubernetes orchestrator without any of the complexities.

You can see an overview of the Azure Container Apps architecture in the following diagram:

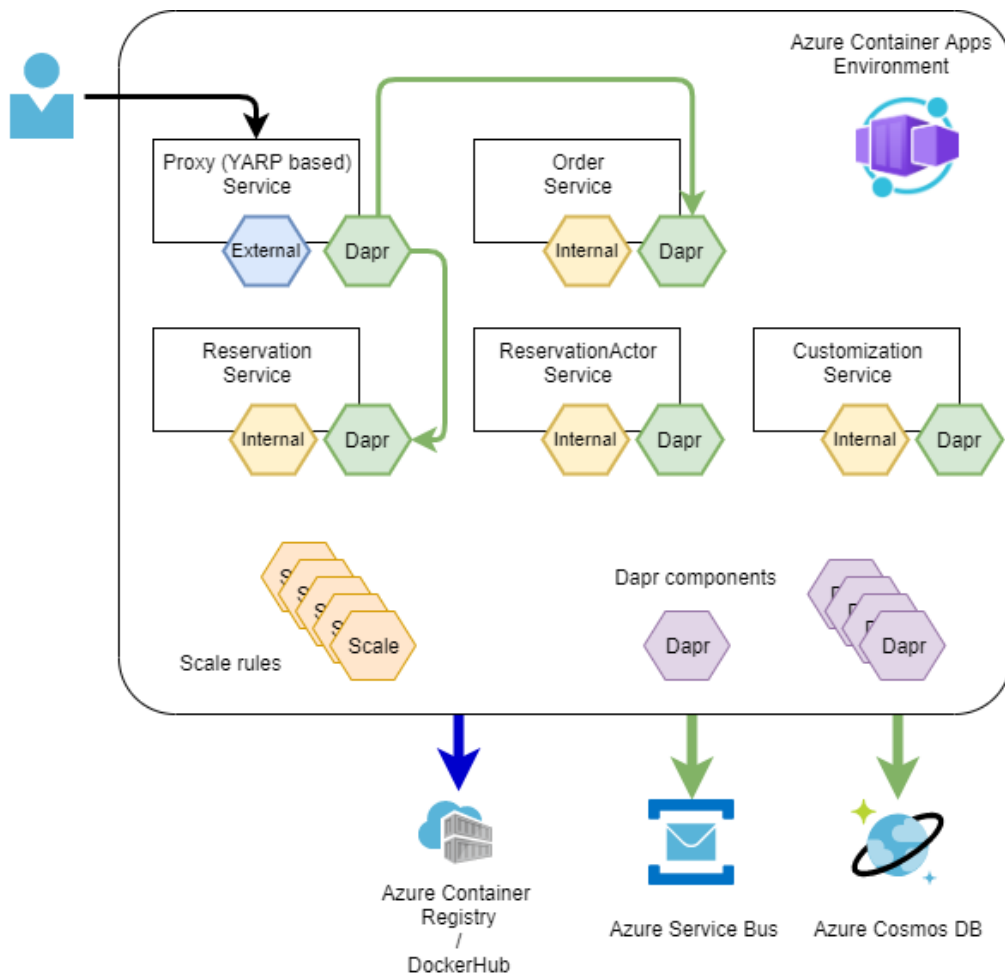


Figure 13.2 – An overall architecture of Azure Container Apps

Hopefully, *Figure 13.2* will not spoil too much the discovery of Azure Container Apps over the next sections. At a very high level, we will find a way to expose some of the solution APIs to external clients; all the involved microservices will leverage the Dapr feature in Azure Container Apps. We will define Dapr components in the Azure Container Apps environment and define which of the Dapr applications can use which ones. Ultimately, we will define scale rules for each Azure container app in the environment.

In the next section, we will start learning how to provision and configure Azure Container Apps.

## Setting up Azure Container Apps

We are going to use the Azure CLI and the Azure portal to manage the deployment of our solution to Azure Container Apps. Let's start with enabling the CLI.

After we log in to Azure, our first objective is to make sure the extension for the CLI and the Azure resource provider are both registered. We can achieve this with the following commands:

```
C:\Repos\practical-dapr\chapter13> az extension add --name
containerapp --upgrade
... omitted ...
C:\Repos\practical-dapr\chapter13> az provider register --
namespace Microsoft.App
... omitted ...
```

Our next steps are set out here:

1. Create a resource group
2. Create a Log Analytics workspace
3. Create an Application Insights resource
4. Create an Azure Container Apps environment

Instructions for the first three steps are provided in the `prepare.ps1` file, using the CLI.

At this stage in our journey, you have probably accumulated a certain level of experience with Azure: therefore, I leave to you the choice of whether to follow the Azure documentation or the provided script to provision the Log Analytics and Application Insights resources via the portal or the CLI.

Let's focus now on the most important step—creating our first Azure Container Apps environment, as you can see in the following command snippet from the `deploy-aca.ps1` file:

```
PS C:\Repos\practical-dapr\chapter13> az containerapp env
create --name $CONTAINERAPPS_ENVIRONMENT --resource-group
$RESOURCE_GROUP --dapr-instrumentation-key
```

```

$APPLICATIONINSIGHTS_KEY --logs-workspace-id
$log_analytics_workspace_client_id --logs-workspace-key
$log_analytics_workspace_client_secret --location $LOCATION
Command group 'containerapp' is in preview and under
development. Reference and support levels:
https://aka.ms/CLI_refstatus
\ Running ..
Container Apps environment created. To deploy a container
app, use: az containerapp create -help
... omitted ...

```

In the previous command snippet, we created an environment and connected it to the Log Analytics workspace. This way, all output and error messages from the container apps will flow there. We also specified which Application Insights instance to use in the Dapr observability building block.

The environment in Azure Container Apps plays a role similar to a cluster in AKS. We are now ready to deploy and configure our microservices as Dapr applications in it.

## Deploying Dapr with Azure Container Apps

In this section, we will learn how to configure Dapr components in Azure Container Apps and how to deploy Dapr applications. These are our next steps:

- Configuring Dapr components in Azure Container Apps
- Exposing Azure Container Apps to external clients
- Observing Azure Container Apps

We start by configuring the components needed by the Dapr applications.

### Configuring Dapr components in Azure Container Apps

In the `components` folder, I have prepared all the components used by the Dapr applications: `reservation-service`, `reservationactor-service`, `customization-service`, and `order-service`.

Let's observe the component in the `components\component-pubsub.yaml` file, used by all microservices to communicate via the **publish and subscribe (pub/sub)** building block of Dapr:

```

name: commonpubsub
type: pubsub.azure.servicebus
version: v1

```

```
metadata:
- name: connectionString
  secretRef: pubsub-servicebus-connectionstring
  value : ""
scopes:
- order-service
- reservation-service
- reservationactor-service
- customization-service
secrets:
- name: pubsub-servicebus-connectionstring
  value: SECRET
```

If you compare the `.yaml` file with the format that we used in *Chapter 6, Publish and Subscribe*, you will notice the terminology used is similar for name and type, but the `.yaml` file is structured differently. As you can read in the documentation at <https://docs.microsoft.com/en-us/azure/container-apps/dapr-overview?tabs=bicep1%2Cyaml#configure-dapr-components>, this is a change specific to Container Apps.

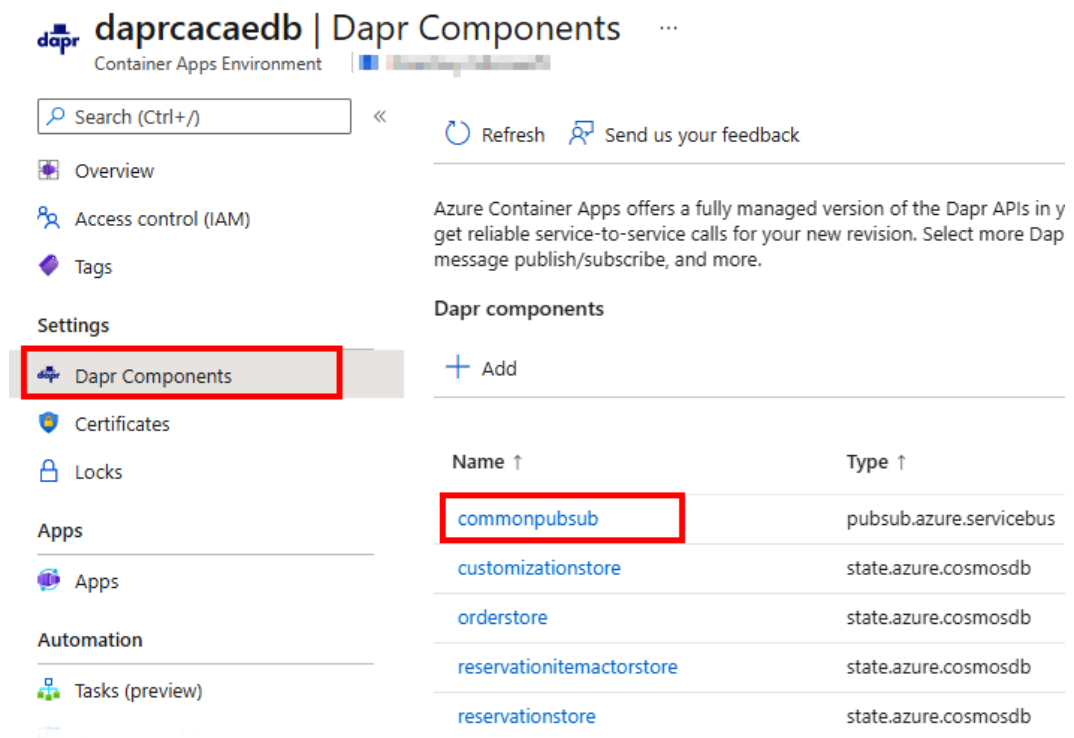
Once you have replaced all the secrets in the components prepared for you in the `components` folder, we can create components via the CLI.

In the `deploy-aca.ps1` file, you can find the commands to create Dapr components, as shown here:

```
PS C:\Repos\practical-dapr\chapter13> az containerapp env
dapr-component set '
--name $CONTAINERAPPS_ENVIRONMENT --resource-group
$RESOURCE_GROUP '
--dapr-component-name commonpubsub '
--yaml ($COMPONENT_PATH + "component-pubsub.yaml")
... omitted ...
```

Once you execute all the commands for the five components, you will have configured the state stores for each Dapr application and the common pub/sub.

We can see the installed components via the CLI or the portal, which offers a great experience with Azure Container Apps:



The screenshot shows the Azure Portal interface for a Dapr environment named 'daprcacaedb'. The left sidebar contains navigation options: Overview, Access control (IAM), Tags, Settings, Dapr Components (highlighted with a red box), Certificates, Locks, Apps, and Automation. The main content area is titled 'Dapr Components' and includes a search bar, a refresh button, and a feedback link. Below this, there is a section for 'Dapr components' with an 'Add' button. A table lists the installed components:

Name ↑	Type ↑
<b>commonpubsub</b>	pubsub.azure.servicebus
customizationstore	state.azure.cosmosdb
orderstore	state.azure.cosmosdb
reservationitemactorstore	state.azure.cosmosdb
reservationstore	state.azure.cosmosdb

Figure 13.3 – Dapr components in an Azure Container Apps environment

In *Figure 13.3*, we can see that by selecting the **Dapr Components** setting in the environment, we get the list of the five components. Let's select the `commonpubsub` component to see its configuration:

## Edit Dapr component

Dapr component details

Name \*

commonpubsub

Component type \* ⓘ

pubsub.azure.servicebus

Version \*

v1

Scopes

Dapr scopes ensure your component is only loaded by applications with specific Dapr App Ids at runtime.

+ Add

App Id	App Name	Delete
order-service ▾	t1-order	
reservation-serv... ▾	t1-reservation	
reservationacto... ▾	t1-reservationactor	
customization-s... ▾	t1-customization	

Secrets

+ Add

Name	Value	Delete
pubsub-servicebus-...	Endpoint=sb://dap...	

Metadata

+ Add

Name	Source	Value	Delete
connectionString	Reference a se... ▾	pubsub-service... ▾	

Figure 13.4 – Settings of a Dapr component in an Azure Container Apps environment

In *Figure 13.4*, we can see the settings of the `commonpubsub` component that we specified via the CLI in the previous steps: they should look very familiar to you as they reflect the elements of a pub/sub Dapr component.

Our next goal is to deploy the Azure Container App to our environment. From the file, let's examine one of the commands:

```
az containerapp create '
  --name t1-reservationactor '
  --resource-group $RESOURCE_GROUP '
  --environment $CONTAINERAPPS_ENVIRONMENT '
  --image ($REGISTRY_NAME + "/sample.microservice
.reservationactor:2.0") '
  --target-port 80 '
  --ingress 'internal' '
  --min-replicas 1 '
  --max-replicas 1 '
  --enable-dapr '
  --dapr-app-port 80 '
  --dapr-app-id reservationactor-service
```

Many elements are noteworthy in the previous `az containerapp create` command. With the `enable-dapr` parameter, we request to onboard Dapr in this application; with `dapr-app-port`, we control the internal port Dapr can use to reach our microservice; and with `dapr-app-id`, we name our microservice in Dapr. With `ingress 'internal'`, we indicate our microservice can be invoked by other microservices in the same environment.

Finally, the `image` parameter gives us the chance to specify the full container image to be used. It can be from an Azure container registry or any other registry such as Docker Hub. As introduced in *Chapter 9, Deploying to Kubernetes*, you can find ready-to-use images in Docker Hub. For instance, the image aligned with this edition of the book for the `reservationactor-service` microservice would be `davidebedin/sample.microservice.reservationactor:2.0`.

By being able to use the very same container image we deployed to a fully managed AKS, we are proving one of the distinctive traits of Azure Container Apps. We are using the same deployment assets, with no changes required to code, to deploy our solution to an environment with capabilities aligned with Kubernetes but with none of its complexities.

Once we have deployed `reservationactor-service`, please proceed to deploy `reservation-service`, `customization-service`, and `order-service`, as shown in the following screenshot. The instructions are available in the `deploy-aca.ps1` file:

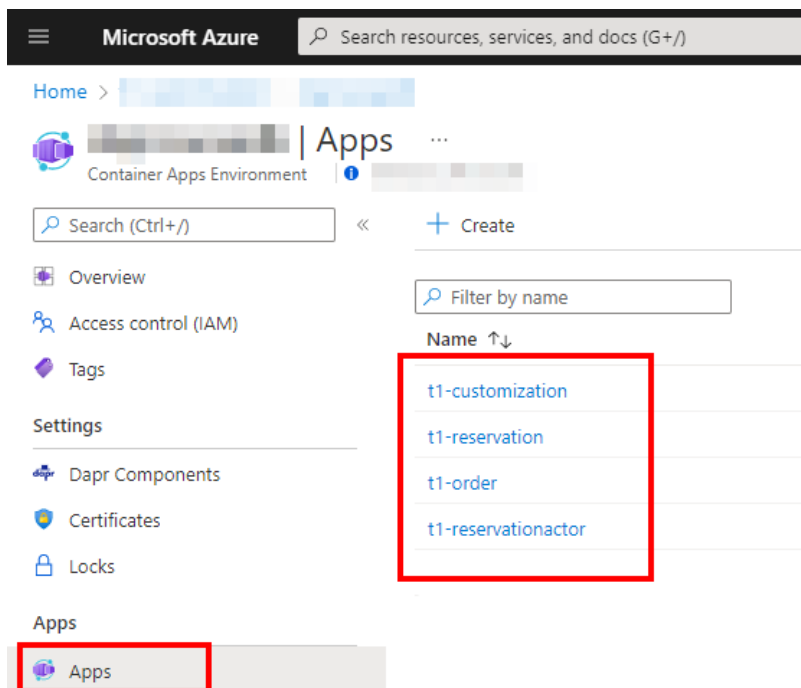


Figure 13.5 – Apps listed in an Azure Container Apps environment

As shown in *Figure 13.5*, if the deployment of the four container apps succeeds, we should be able to see these in the **Apps** section of the environment resource using the Azure portal.

How can we now test our deployed application? Let's explore a new option to expose a container app to external clients.

## Exposing Azure container apps to external clients

If we examine the configuration for all of our microservices as Dapr applications, each one has `ingress` as `internal`: they can communicate with each other, but they are not reachable from external clients. Is something missing here?

In *Chapter 10, Exposing Dapr Applications*, we learned how to use reverse proxies such as NGINX and an API manager such as Azure API Management to expose our solution to an external audience.

We could apply any of these two approaches to deploy an abstraction layer between our Dapr applications and the externally reachable API. Instead, I took the chance to propose a different approach for a nimble reverse proxy built as a Dapr application with **Yet Another Reverse Proxy (YARP)** (documentation at <https://microsoft.github.io/reverse-proxy/>): an open source project for an easy-to-use, yet flexible reverse proxy built with .NET.

Following the very same approach we used to create projects for the first Dapr applications in *Chapter 4, Service-to-Service Invocation*, I prepared a fully functional project in the `\sample.proxy` folder, with a corresponding Docker file in case you want to build it or a ready-to-use `davidebedin/sample.proxy:0.1.6` image available on Docker Hub.

A YARP reverse proxy can be configured via code or the `appsettings.json` file of the .NET project. You can find the rules at `sample.proxy\appsettings.json`. As an example of the configuration, let's observe the following snippet:

```
{
  ... omitted ...
  "AllowedHosts": "*",
  "ReverseProxy": {
    "Routes": {
      ... omitted ...
      "balance" : {
        "ClusterId": "daprsidecar",
        "Match": {
          "Path": "/balance/{**remainder}",
          "Methods" : [ "GET" ]
        },
      },
      "Transforms": [
        { "PathPattern": "invoke/reservation-
          service/method/balance/{**remainder}" }
      ]
    }
  },
  "Clusters": {
    "daprsidecar": {
      "Destinations": {
        "dapr": {
          "Address": "http://localhost:{{DAPR_HTTP_PORT}}
            /v1.0/"
        }
      }
    }
  }
}
```

```
}
}
```

The preceding configuration shows a simple rule to accept a route with the path as `/balance/{**remainder}`, in which `{**remainder}` identifies anything passed afterward. The requested path gets transformed to adhere to the Dapr methods for the sidecar, resulting in a `invoke/reservation-service/method/balance/{**remainder}` path. The route, then transformed, is sent to a cluster backend, in this case with the Dapr sidecar as a destination.

It is very simple to deploy a YARP-based reverse proxy such as this `sample.proxy` project as a Dapr application in Azure Container Apps. In the `deploy-aca.ps1` file, we find the following command:

```
az containerapp create '
--name tl-proxy '
--resource-group $RESOURCE_GROUP '
--environment $CONTAINERAPPS_ENVIRONMENT '
--image ($REGISTRY_NAME + "/sample.proxy:0.1.6") '
--target-port 80 '
--ingress 'external' '
--min-replicas 1 '
--max-replicas 1 '
--enable-dapr '
--dapr-app-port 80 '
--dapr-app-id proxy-service
```

What has changed in this Dapr-enabled container app, compared to the previous ones, is that the `ingress` parameter is now set to `external`.

We can also verify this configuration via the Azure portal, as shown in the following screenshot, which shows the **Ingress** settings for the container app:

Enable ingress for applications that need an HTTP endpoint.

HTTP Ingress ⓘ ☐ Disabled ☒ Enabled

Ingress traffic ☐ Limited to Container Apps Environment ☐ Limited to VNet: Applies if internalOnly is set to true on the Container Apps environment ☒ Accepting traffic from anywhere: Applies if internalOnly is set to false on the Container Apps environment

Target port ⓘ

Figure 13.6 – Ingress configuration of a container app

Azure Container Apps will provide an externally reachable address that we can get in the portal or via the CLI, as shown in the following command:

```
PS C:\Repos\practical-dapr\chapter13> (az containerapp show
--resource-group $RESOURCE_GROUP --name t1-proxy --query
"properties.configuration.ingress.fqdn" -o tsv)
t1-proxy.OMITTED.azurecontainerapps.io
```

A layer of indirection between clients and our APIs is always important as it does give us ample options for evolution, especially in a microservices-oriented architecture. A proper, fully-featured API manager such as Azure API Management might be a better choice in some scenarios. In our case, with the objective to simplify the deployment of our solution and offer it as a service to external clients, a simple and effective reverse proxy such as YARP can be a good match.

At this stage, all five Azure container apps have been deployed. Let's check this with the Azure CLI, with the following command taken from the `deploy-aca.ps1` file:

```
PS C:\Repos\practical-dapr\chapter13> az containerapp
list '
--resource-group $RESOURCE_GROUP '
--query "[].{Name:name, Provisioned:properties
.provisioningState}'
-o table
Name                                Provisioned
-----
t1-customization                    Succeeded
t1-order                            Succeeded
t1-proxy                            Succeeded
t1-reservationactor                 Succeeded
t1-reservation                      Succeeded
```

Our next step is to test our solution deployed to Azure Container Apps and observe its behavior.

## Observing Azure container apps

Azure Container Apps offers several tools and approaches to monitor the health and behavior of our applications, as listed here:

- Log streaming
- Application Insights

- Logs in Log Analytics
- Container console
- Azure Monitor

Let's examine the first two, log streaming and Application Insights:

- **Log streaming:** This is the equivalent of the `kubectl` command we learned to use in *Chapter 9, Deploying to Kubernetes*. This time, we can read the output of the log coming from all the instances of a container in a specific Azure container app in the Azure portal or via the Azure CLI. You can learn more at <https://docs.microsoft.com/en-us/azure/container-apps/observability?tabs=powershell#log-streaming>. With log streaming, you get a stream of real-time logs from the container app `stdout` and `stderr` output.
- **Application Insights:** This is an observability option for Dapr that we only mentioned in the previous chapters. With Azure Container Apps in conjunction with Dapr, it is the primary **end-to-end (E2E)** observability option.

Before we learn how to leverage these two options, let's run a test of the overall solution, which, in turn, will generate log messages and metrics we can observe. In the `test.http` file, we have our API calls submitting an order and retrieving the resulting quantity balance for a product involved in the overall process, from order to customization:

```
@baseUrl = x.y.z.k
###
GET https://{baseUrl}/balance/bussola1
###
# NON problematic order
# @name simpleOrderRequest
POST https://{baseUrl}/order HTTP/1.1
content-type: application/json
{
  "CustomerCode": "Davide",
  "Date": "{{datetime 'YYYY-MM-DD'}}",
  "Items": [
    ... omitted ...
  ],
  "SpecialRequests" : []
}
... continued ...
```

In the snippet from `test.http`, you have to replace the value of the `@baseUrl` variable with the externally reachable **Fully Qualified Domain Name (FQDN)** of the Azure container app named `t1-proxy`, which acts as the external endpoint for the solution: we obtained it via the CLI in the previous section; otherwise, you can read it from the Azure portal.

If we succeeded in the configuration and deployment of our solution in Azure Container Apps, we should receive a positive result with the balance of a **stock-keeping unit (SKU)** (a product), and our order should have been accepted. If so, it means the microservices of our solution successfully managed to interact via Dapr.

It is time to first observe the `t1-proxy` container app. In the Azure portal, let's select the **Log Stream** option from the **Monitoring** section. As depicted in *Figure 13.7*, we have the choice to observe the container for the application or the Dapr sidecar container:

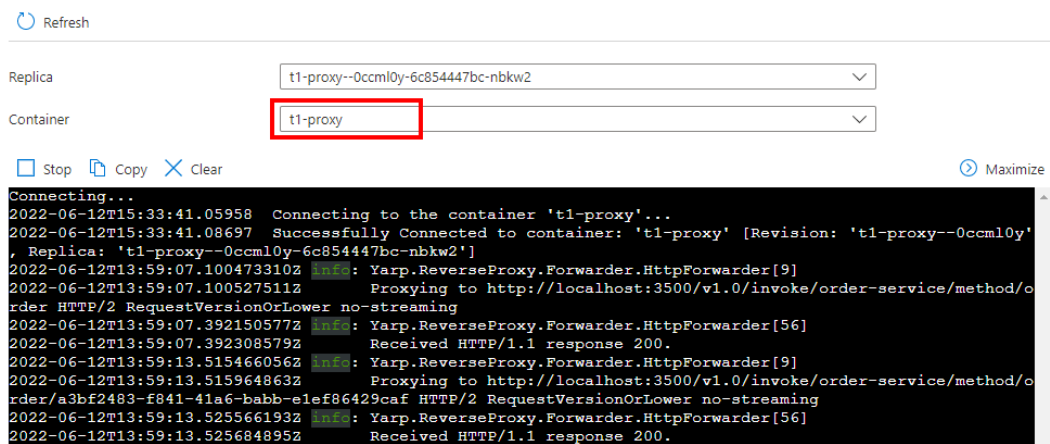


Figure 13.7 – Log stream of the `t1-proxy` container app

As we can observe in the preceding screenshot, we see the logs confirming our requests reached the YARP-based reverse proxy and have been processed by interacting with the local Dapr sidecar.

As getting balance or posting order route calls both interact with the `reservationactor-service` microservice running in the `t1-reservationactor` container app, we can observe interactions reaching this microservice via Dapr service-to-service and actor building blocks by looking at the output in its log stream, as follows:

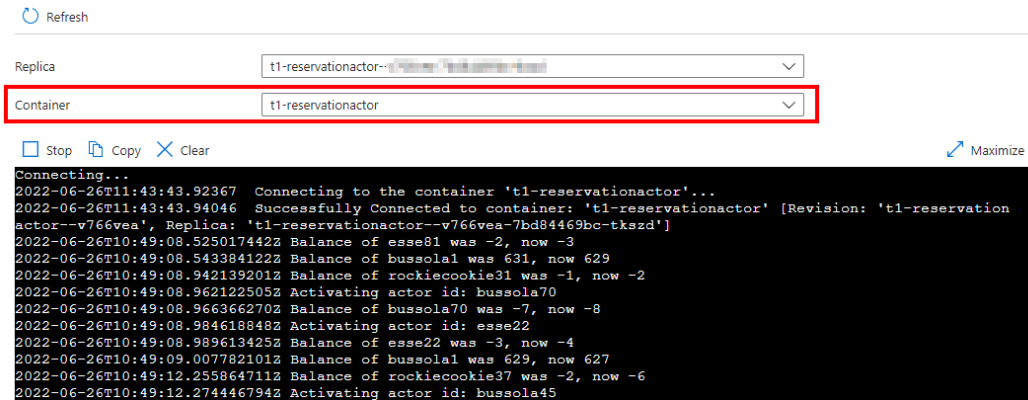


Figure 13.8 – Log stream of the `t1-reservationactor` container app

As shown in *Figure 13.8*, with Azure Container Apps, we can have the same visibility on the output from a microservice that we have locally in our development environment, all with the simplicity of the Azure portal or CLI.

Another observability option in Azure Container Apps is Application Insights. While provisioning the container app environment with `az containerapp env create ... --dapr-instrumentation-key`, we specified an Application Insights instance to be configured for the Dapr observability building block.

If we inspect the **Application map** section in the Azure Application Insights resource, we should see a representation of interactions between our Dapr microservices, like the one in *Figure 13.9*:

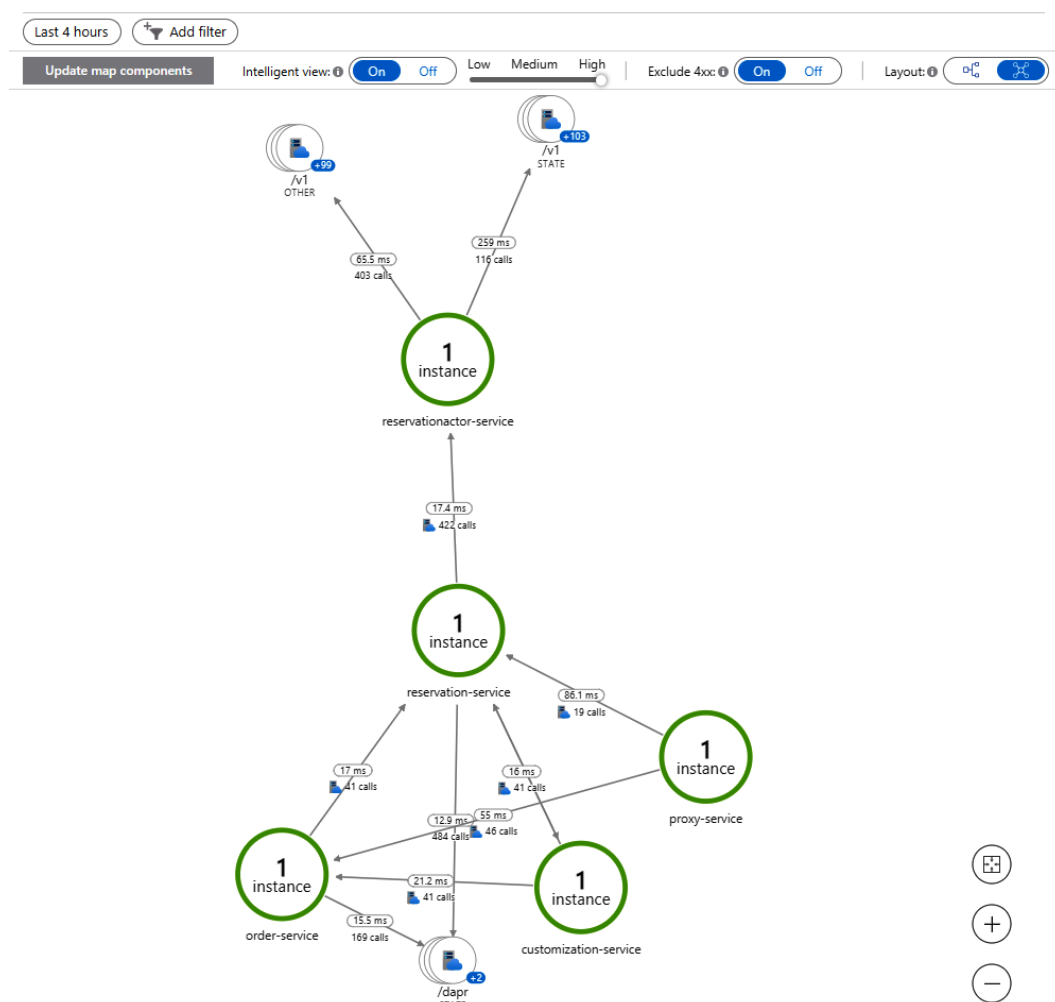


Figure 13.9 – Application Insights application map

In *Chapter 11, Tracing Dapr Applications*, we explored the several options Dapr provides us for observing the interactions between microservices: in the deployment of our solution to AKS, we explored Zipkin, while Application Insights was an option we did not explore. In Azure Container Apps, Application Insights is the primary observability option for Dapr microservices.

In the next section, we will explore a powerful feature of Azure Container Apps to scale workload based on load.

## Autoscaling Azure Container Apps with KEDA

In this section, we will learn how to leverage the autoscale features of Azure Container Apps. We will cover the following main topics:

- Learning about KEDA autoscalers
- Applying KEDA to Azure container apps
- Testing KEDA with container apps

Our first objective is to learn about KEDA.

### Learning about KEDA autoscalers

The Azure Container Apps service provides a powerful abstraction on a complex topic such as scaling microservices by abstracting the Kubernetes default scaling concepts and by adopting **KEDA**. You can learn all about it in the documentation at <https://docs.microsoft.com/en-us/azure/container-apps/scale-app>.

KEDA is an open source project started by Red Hat and Microsoft, now a CNCF incubating project, that extends the capabilities of the Kubernetes autoscaler to allow developers to scale workload on event-based metrics coming from an incredibly diverse set of scalers. As an example, with KEDA, you can scale your application component based on the number of pending messages in an Azure Service Bus subscription. You can find more details about KEDA at <https://keda.sh/>.

In *Chapter 12, Load Testing and Scaling Dapr*, we experimented with the **Locust** open source testing framework to generate, relying on Azure resources, an increasing volume of requests on our applications. The testing framework created simulated a pattern of requests against the APIs to measure the response of the infrastructure and applications under load and to verify how the scaling behavior in an AKS deployment could help to achieve optimal performance.

In *Chapter 12*, we also specified scaling rules leveraging the default metrics available in any **Kubernetes** autoscaler: usage of CPU and memory. Those metrics are an effective scaling indicator for memory and CPU-intensive workload.

It is worth reminding you that, by leveraging Dapr, most data operations occur outside of the microservices in our solution. The Dapr sidecar performs **input/output (I/O)** operations toward Dapr components, relying on **Platform as a Service (PaaS)** services such as Azure Cosmos DB for managing the state of each microservice, and asynchronous communications between microservices are supported by Azure Service Bus.

Scaling up and down the instances of a microservice based on the number of pending messages in a queue is a far more effective approach to anticipating the stress conditions than measuring the impact of too many messages reaching too few instances. KEDA is a powerful approach to anticipating, rather than reacting to, stress conditions common to **event-driven architecture (EDA)**.

Azure Container Apps leverages many open source technologies including KEDA, which enables the user. With KEDA, we can configure scaling rules on external resources; for example, we can scale a container app based on the pending messages in an Azure Service Bus subscription.

As we have learned the basics of KEDA autoscalers, it is time to apply these configurations to each of our container apps.

## Applying KEDA to Azure container apps

Our first objective is to configure scale rules for our container apps. Secondly, we will launch a load test with Locust to verify if the container apps scale accordingly to plan.

In the `\scale` folder, I prepared `.yaml` files with scale rules for each of the five container apps. We will configure these via the Azure CLI with the code prepared in `scale-aca.ps1`.

For instance, let's see the content of the `scale\scale-reservationservice.yaml` file, which is going to be applied to the `reservation-service` microservice configuration:

```
properties:
  template:
    scale:
      minReplicas: 1
      maxReplicas: 5
      rules:
        - name: servicebus
          custom:
            type: azure-servicebus
          metadata:
            topicName: onorder_submitted
            subscriptionName: reservation-service
            messageCount: 10
      auth:
        - secretRef: pubsub-secret
          triggerParameter: connection
```

In the previous snippet, we configured the container app to scale between 1 and 5 replicas, following a custom rule using the KEDA scaler of type `azure-servicebus`. This instructs the KEDA component in Azure Container Apps to add an instance if there are more than 10 pending messages, as specified with `messageCount` in the `subscriptionName` of the `topicName`. You can also notice a secret named `pubsub-secret` is referenced.

In *Chapter 6, Publish and Subscribe*, we learned how Dapr can leverage Azure Service Bus for messaging between microservices. While exploring our sample solution, Dapr automatically created several topics and subscriptions in Azure Service Bus. KEDA will observe these subscription metrics to understand if the microservice is processing messages as fast as possible or is in need of extra help by an additional instance.

In addition to KEDA, Azure Container Apps provides CPU, memory, and HTTP scale rules. Let's examine the following `scale\scale-proxy.yaml` file prepared for the `proxy-service` microservice:

```
properties:
  template:
    scale:
      minReplicas: 1
      maxReplicas: 4
      rules:
        - name: httpscale
          http:
            metadata:
              concurrentRequests: '10'
        - name: cpuscalerule
          custom:
            type: cpu
            metadata:
              type: averagevalue
              value: '50'
```

In the previous snippet, we ask to scale instances, from 1 and up to 4, if there are more than 10 concurrent HTTP requests or the CPU exceeds 50% on average.

How can we apply these new scale rules to the configurations of our deployed container apps? We will use the Azure CLI to create a new revision for each container app. As an example, let's examine a snippet from `scale-aca.ps1`:

```
az containerapp secret set '
--name tl-reservation '
```

```
--resource-group $RESOURCE_GROUP '  
--secrets pubsub-secret=$SB_CONNECTION_STRING  
... omitted ...  
az containerapp revision copy '  
--name t1-reservation '  
--resource-group $RESOURCE_GROUP '  
--yaml .\scale\scale-reservationservice.yaml
```

In the previous code snippet, we are first creating a secret to keep the connection string to the Azure Service Bus resource, and secondly, we are creating a new revision for the `t1-reservation` container apps, specifying the `scale\scale-reservationservice.yaml` file for all the configurations.

Having executed the command, a new revision is going to be created.

### Revision management

According to the default single revision mode, only one revision is active at the same time. The new one takes the place of the old revision.

Azure Container Apps can support more complex scenarios. In multiple revision mode, for example, you can split traffic between the existing revision and the new one, both active. The traffic splitting of Azure Container Apps can be instructed to direct a small portion of traffic to the revision carrying the updated application, while most requests continue to be served by the previous revision.

As it is common with canary deployment practices, if the newer revision properly behaves, the traffic can be increasingly shifted toward the newer revision. Finally, the previous revision can be disabled.

You can find more information about revision management in Container Apps in the documentation at <https://docs.microsoft.com/en-us/azure/container-apps/revisions>.

Let's examine a scale rule on the Azure portal, as shown in the following screenshot.

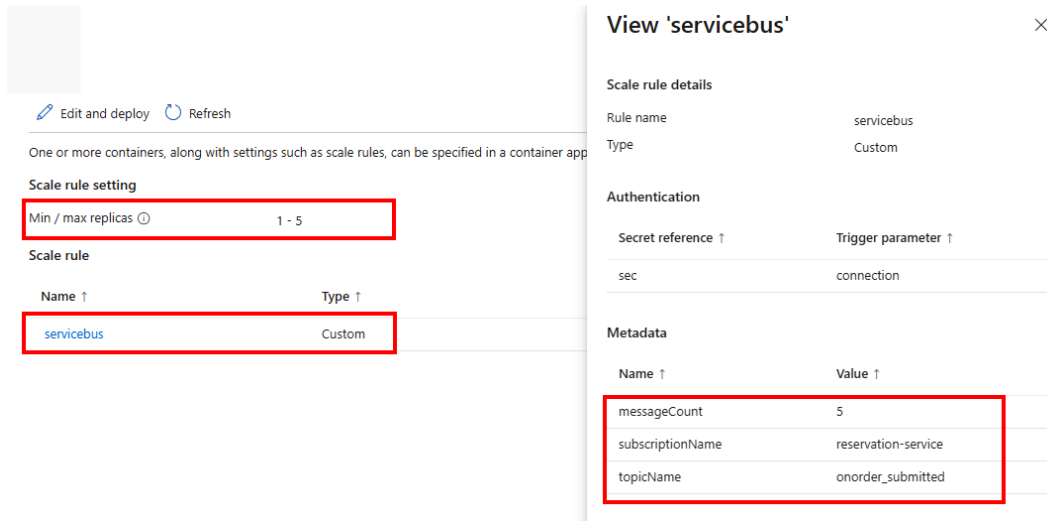


Figure 13.10 – KEDA scale rule applied to a container app

As shown in *Figure 13.10*, the **Scale** setting for the container app named `t1-reservation`, which supports the `reservation-service` Dapr application, shows the scale rule we configured with the KEDA scaler.

If we successfully executed all the commands in `scale-aca.ps1`, we should have new revisions for all our five container apps, some with scale rules reacting to pending messages in an Azure Service Bus subscription, and others with scale rules reacting to average used CPU or memory.

Our next step is to run a load test on our solution deployed to Azure Container Apps to test if the scale rules have any impact on the number of instances.

## Testing KEDA with container apps

We will reuse the knowledge gathered in *Chapter 12, Load Testing and Scaling Dapr*, to run a load test with the Locust load testing framework.

If you forgot the syntax, I suggest you go back to the *Chapter 12* instructions to refresh your knowledge of Locust and the distributed testing infrastructure leveraging Azure resources.

In the scope of this chapter, the load test can be the very same available in *Chapter 12* or the test provided in the `loadtest\locustfile.py` file. The main change we need to apply to the Locust test, whether we are launching it via the command line or accessing the web UI deployed to Azure, is to replace the hostname with the URL of the ingress to the externally exposed `t1-proxy` container

app, which can also be gathered with the CLI command, as previously described in this chapter. The rest of the test is the same.

The load test is designed to request balance and submit orders, randomly picking products (also known as SKUs) from a previously defined set. If you are reusing the state stores from *Chapter 12* also in this chapter, you probably already have a range of products available. On the contrary, if you reset the data or provisioned new state stores, in the `loader\generatedata.py` file there is a Python script to provision the SKU/products so that the load test can find it.

With the following command, you can regenerate the state for all the SKUs you need in the tests, as the script leverages a route supported by `proxy-service` to refill the balance of SKUs in the catalog:

```
PS C:\Repos\practical-dapr\chapter13> python
.\loader\generatedata.py https://<BASE_URL>
    SKU:cookie000, Balance:
    {"sku":"cookie000","balanceQuantity":63}
    SKU:cookie001, Balance:
    {"sku":"cookie001","balanceQuantity":32}
    ... omitted ...
```

After you have made sure there are plenty of products in the configured state stores, we can proceed with the load test.

Please consider that there are many factors influencing the outcome of a load test, especially the capacity of the state store's backend (Azure Cosmos DB) and the message bus (the Azure Service Bus resource). Luckily for us, these cloud services can reach the performance objectives if we properly configure the service-level tier.

Given this preparation and context, let's launch a Locust load test and see if the Azure Container Apps instances scale as expected.

One way to get a comprehensive view of instance count for each of the five is to compose a dashboard in the Azure portal displaying the **Replica Count** metric, accessible from the **Metrics** sidebar menu in the **Monitoring** section of an Azure container app page, as illustrated in the following screenshot (for more information on how to compose dashboards in the Azure portal, please see the documentation at <https://docs.microsoft.com/en-us/azure/azure-portal/azure-portal-dashboards>. For a deep dive into Azure Container Apps monitoring, please check <https://docs.microsoft.com/en-us/azure/container-apps/observability?tabs=bash#azure-monitor-metrics>):

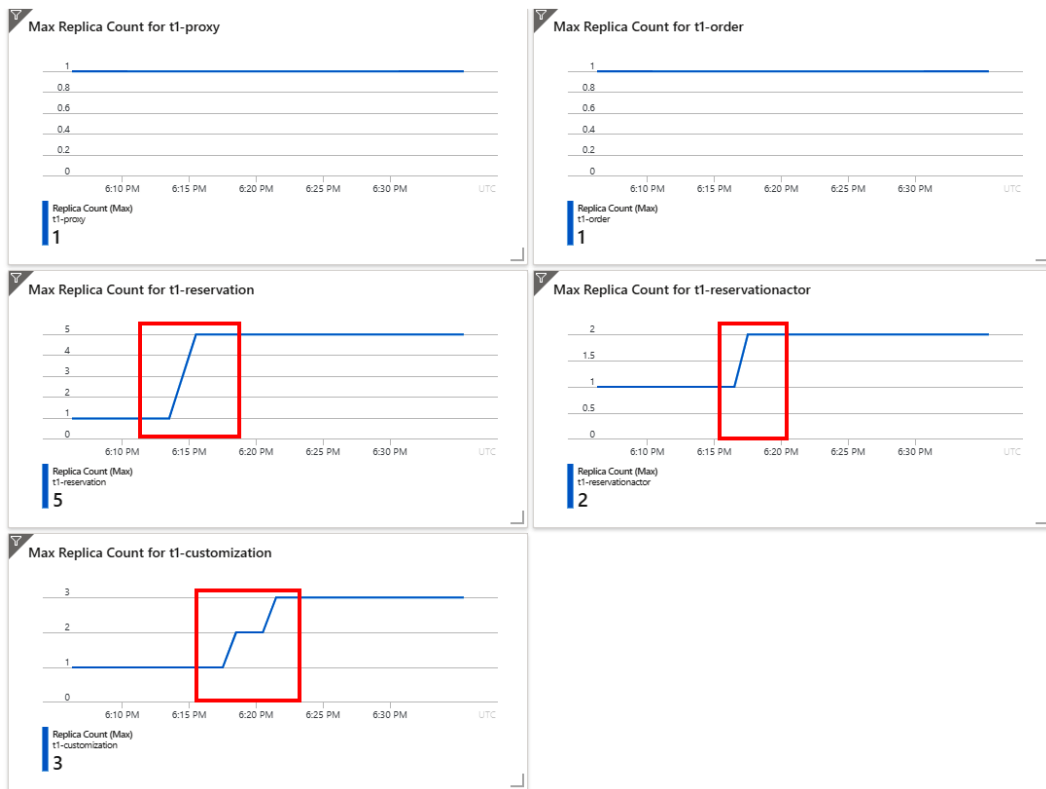


Figure 13.11 – Azure Container Apps instances scaling up

In *Figure 13.11*, we can see that `t1-reservation`, `t1-customization`, and `t1-reservationactor` container apps are scaling up.

The first two container apps, corresponding to `reservation-service` and `customization-service` Dapr microservices, have KEDA scale rules triggered by the increasing number of pending messages in their respective Azure Service Bus subscriptions.

The latest, corresponding to the `reservationactor-service` Dapr application, is scaling based on the overall memory usage. As this microservice is leveraging the actor building block of Dapr, memory could be a good indicator of growth in demand: the more actors kept alive, the more memory needed.

In this section, we learned how to create new revisions of Azure container apps with the intent of setting up scale rules, using default and KEDA scalers.

## Summary

In this chapter, we learned the core concepts of Azure Container Apps, the newest serverless container option in Azure.

We effortlessly managed to deploy the Dapr microservices to Azure Container Apps with no changes to the code or containers. By leveraging a simplified control plane via the Azure portal and the Azure CLI, we configured a complex solution with minimal effort.

By experimenting with revisions, we briefly touched on a powerful feature of Azure Container Apps that can simplify the adoption of blue/green or canary deployment practices.

By testing the scaling features in Azure Container Apps, we learned how easy it is to leverage KEDA, as it is an integral part of the Azure service.

The objective of the fictional company behind the *Biscotti Brutti Ma Buoni* solution, stated at the beginning of this chapter, was to find a suitable platform to offer the solution—the one we built with Dapr in this book—to external businesses with a SaaS approach.

While the fictional company had experience with Kubernetes, a serverless platform such as Azure Container Apps will greatly simplify the management of their solution, providing a scalable managed environment with built-in support for Dapr.

With this chapter, we complete our journey in learning Dapr through the development, configuration, and deployment of a microservices-oriented architecture.

There are many topics to be explored in Dapr at <https://docs.dapr.io/>. As we learned, Dapr is a rapidly growing open source project with a rich roadmap (see <https://docs.dapr.io/contributing/roadmap/>), and lots of interesting new features are planned for the next versions of Dapr.

I hope you found this book interesting, that you learned the basics of Dapr, and that it stimulated you to learn more.

# Assessments

This section contains answers to questions from all chapters.

## Chapter 1, Introducing Dapr

1. Service-to-service invocation, state management, publish and subscribe, bindings, actors, observability, secrets, and configuration.
2. The CLI lets you interact with the local or remote runtime.
3. First, install Dapr CLI, then initialize Dapr.
4. You can install Dapr on Kubernetes via the CLI or Helm.

## Chapter 2, Debugging Dapr Solutions

1. Using the Dapr extension, you can prepare a project for debugging.
2. You can follow the usual approach to combine the debug configuration of multiple projects together.
3. You can launch the project via launch configuration, using the Dapr CLI or Tye, and then attach VS Code to the corresponding process.
4. You adapt the Tye recipe for Dapr to fit your solution's specifics.

## Chapter 3, Microservices Architecture with Dapr

1. Bounded contexts are so relevant because they help architects in taming the complexity of the overall architecture into more easily manageable microservices.
2. Given that runtimes such as Dapr are available, there is a lower barrier of entry in adopting microservices: nevertheless, the size and culture of your team are more relevant in deciding whether you are up with the task.
3. In my experience, observability is the most underrated requirement in a microservices architecture, which Dapr can bring significant advantages to.

## Chapter 4, Service-to-Service Invocation

1. A lot! A few examples are the minimal hosting model and minimal APIs
2. Dapr does always rely on gRPC for communication between its sidecars.
3. For most microservices, there is not a need for gRPC; if you are looking for network efficiency between hosts/nodes, then Dapr will give you that benefit with service-to-service invocation.
4. It depends on what you value most; I prefer the simplicity of HTTP on the microservice side.

# Index

## Symbols

**.NET 6** 11, 86

installing 10, 11

**.NET solutions**

creating 29

**.NET support policy**

reference link 11

## A

**actors, using in Dapr** 136

configuration, verifying 141, 142

state store, configuring 140, 141

virtual actor pattern 136-140

**AKS cluster**

connecting to 161, 162

creating 161

**Amazon Web Services (AWS)** 4

**API management**

setting up, on Kubernetes 189-195

used, for configuring Dapr policies 195-203

**Application Insights** 263

**application programming interface**

(API) 6, 41, 100, 145, 218

**ASP.NET service** 86

**autonomous microservices** 48

**Azure API Management (APIM)** 7, 189, 212

**Azure Container Apps** 7

architecture 248-250

autoscaling, with KEDA 267

components 248

concepts 250

Dapr components, configuring 254-258

Dapr, deploying with 254

exposing, to external clients 259-262

functionalities 249

KEDA, applying 268-271

KEDA, testing with 271-273

need for 250-253

observing 262-266

setting up 253, 254

tools and approaches 263

**Azure Container Instances (ACI)** 234

**Azure Container Registry (ACR)** 170

**Azure Cosmos DB** 89

partitioning with 95-97

setting up 89, 90

state store, configuring 90-93

state store, testing 93-95

using, as state store 89

using, scenarios 97, 98

**Azure Event Hubs input binding**

- creating 129, 130
- configuring 130, 131
- events, producing 132, 133
- implementing 131, 132
- used, for ingesting data 128, 129

**Azure Key Vault 172****Azure Kubernetes Service (AKS) 7, 248**

- reference link 230

**Azure resource group**

- creating 160

**Azure Service Bus (ASB) 217**

- using, in Dapr 102, 103

**B****Bash 226****bounded context 42****bounded contexts, e-commerce architecture**

- manufacturing 47
- sales 46
- shipping 47

**business continuity (BC) 251****C****C# 199****central processing unit (CPU) 206****Certificate Authority (CA) 8****certification life cycle**

- reference link 81

**circuit breaker 58**

- reference link 58

**CloudEvents 111****Cloud Native Computing Foundation (CNCF) 4, 111, 208, 245, 248****Cluster Autoscaler (CA) 230****command-line interface (CLI) 21, 104****compound debug sessions**

- launching 33, 34

**container apps 250****container registry 170****container repository 170****Continuous Integration/Continuous Deployment (CI/CD) pipeline 13****cron binding**

- testing 122

**cron input binding**

- configuring 121

**D****Dapr actor model**

- concurrency 143, 144
- consistency 144
- lifetime 145
- placement service 142, 143
- resiliency 145

**Dapr API 7****Dapr application**

- deploying 174-179
- deploying, to Kubernetes 166
- exposing, to external clients 179-184

**Dapr bindings**

- reference link 121
- using 120, 121

**Dapr CLI 7**

- installing 10
- reference link 10

**Dapr C# SDK 82****Dapr debug, in VS Code**

- configuring 22
- debug configuration, examining 24-28
- debugger, attaching 22-24

**Dapr documentation**

- reference link 9

**Dapr Helm charts** 7

**Dapr host** 7

**Dapr multi-project solution**

.NET solutions, creating 29

compound debug sessions, launching 33, 34

configuration, launching 30, 31

debugging 28, 29

debug sessions, launching

individually 32, 33

tasks 31, 32

**Dapr operator** 8

**Dapr placement service** 8

**Dapr policies**

configuring, with API management 195-203

**Dapr Roadmap**

reference link 9

**Dapr runtime** 7, 54

**Dapr SDKs**

reference link 6

**Dapr Sentry** 8

**Dapr service invocation API** 54

**Dapr sidecar injector** 8

**Dapr topic**

subscribing to 114, 115

**data**

ingesting, with Azure Event Hubs

input binding 128, 129

**Database as a Service (DBaaS)** 80

**Data Transfer Object (DTO)** 86, 103

**debug sessions**

launching, individually 32, 33

**declaratively** 104

**declarative subscriptions** 104

**Dependency Injection (DI)** 113, 114

**Distributed Application Runtime**

(Dapr) 4, 6, 21, 248

.NET 6, installing 10, 11

actors, using 136

Azure Service Bus (ASB), using 102, 103

building blocks 8, 9

components, configuring in Azure

Container Apps 254-258

data, preparing via port-forwarding 236, 237

deploying, with Azure Container Apps 254

Docker 10

documentation link 140

exploring 7

history 5

Horizontal Pod Autoscaler,

operating 242-245

installing, on Kubernetes 13

installing, on self-hosted mode 12, 13

load testing 236

Locust on Azure Container

Instances 240-242

Locust, testing locally 238-240

messages, publishing to 113

misconceptions, eliminating 6, 7

observability applications 206-208

pub/sub pattern, using 99-102

reference link 211, 231

sample, building 14-18

scaling, on Kubernetes 227-229

services, invoking with 54, 55

setting up 9

setting up, on Kubernetes 162-166

state management 80

tools and components 7, 8

tracing, enabling 211-213

Twilio output bindings, using 122, 123

Tye, using with 35

version, updating 13, 14

VS Code, installing 11

Windows Terminal, installing 11

**Docker**

reference link 10

**Docker Hub** 166, 170

**Docker images**

building 167-170

pushing 170-172

## E

**e-commerce architecture**

bounded contexts 46

building 45

sales microservices example 47

**e-commerce ordering system**

stateful services 85

**e-commerce reservation system**

actors, assigning from other Dapr

applications 150-154

actors, implementing 146

actor's projects, preparing 146, 147

actor's state, inspecting 154, 155

**end-to-end (E2E)** 200, 263

**Envoy** 248

**ETag** 82

**ETAg** 140

**event-driven architecture (EDA)** 42, 43, 268

**event-driven microservices** 49

## F

**Fully Qualified Domain Name (FQDN)** 264

**Function as a Service (FaaS)** 44

## G

**Google Cloud Platform (GCP)** 4, 101

**Grafana** 219, 227, 251

dashboard, importing 221-223

installing 220

used, for analyzing metrics 219

**gRPC microservice, creating** 69

copy of order-service, creating 73, 74

project, configuring 70

proto file, creating 70

reservation-service, creating 69

service implementation 71-73

test integration 74, 75

**gRPC Remote Procedure Call**

(gRPC) 6, 30, 68, 218

in ASP.NET 68

latency, winning with 76

references 68

## H

**Hashicorp Vault** 172

**Helm** 162, 180, 188, 251

**Horizontal Pod Autoscaler (HPA)** 230, 250

**HyperText Transfer Protocol**

(HTTP) 6, 30, 102

## I

**ingress controller (IC)** 180

daprizing 186-189

**input/output (I/O)** 267

**Internet Protocol (IP)** 210

## J

**JavaScript Object Notation (JSON)** 103

## K

**KEDA autoscalers** 267, 268

**Key-Value Pair (KVP)** 8

**kubectrl** 251

**kubectrl CLI** 161

**kubectl port-forward command**

reference link 210

**Kubernetes autoscaler 267****Kubernetes-based Event-Driven****Autoscaling (KEDA) 245, 248**

applying, to Azure Container Apps 268-271

autoscaling with 245

reference link 245

testing, with Azure Container Apps 271-273

used, for autoscaling Azure

Container Apps 267

**Kubernetes, concept**

autoscale 230

replicas 229

resource requests and limits 231-234

**Kubernetes features**

reference link 230

**Kubernetes (k8s) 160**

AKS cluster, creating 161

API management, setting up on 189-195

Azure resource group, creating 160

configuring 227

connecting, to AKS cluster 161, 162

Dapr application, deploying to 166

Dapr, installing 13

Dapr, scaling 227-229

Dapr, setting up on 162-166

secrets, managing 172-174

setting up 160

**L****Language-Integrated Query (LINQ) 117****load testing 234****Locust 227, 234, 267**

reference link 227

URL 234

used, for load testing 234-236

**Locust on Azure 235****log streaming 263****Long-Term Support (LTS) 11****loosely coupled microservices 48****M****managed service provider (MSP) 251****master node 234****mDNS (Multicast DNS) 57****Message Queuing Telemetry****Transport (MQTT) 101****messaging systems**

subset 101

**microservices**

automated deployment 42

autonomy 41

bounded context 42

discovering 40, 41

event-driven architecture 42, 43

loose coupling 42

observability 43

service 41

sustainability 43

**microservices patterns**

adopting 44, 45

benefits 44

evolution 44

flexibility 44

reliability 44

scale 44

**microservices, with Dapr**

autonomous microservices 48

event-driven microservices 49

loosely coupled microservices 48

observable microservices 48

scalable microservices 49

stateless microservices 49

**Microsoft Azure** 4  
**minimal APIs** 64  
    reference link 64  
**Model-View-Controller (MVC) pattern** 15  
**mutual Transport Layer Security (mTLS)** 8

## N

**name resolution** 57  
**Neural Autonomic Transport  
    System (NATS)** 101  
**NGINX** 180  
**NGINX ingress controller** 212

## O

**observable microservices** 48  
**OpenTelemetry** 208  
    URL 208  
**OpenTelemetry Collector** 208  
**operations** 195

## P

**partitioning** 95  
    with Cosmos DB 95-97  
**per-actor locks** 143  
**placement service**  
    reference link 143  
**Platform as a Service (PaaS)** 190, 267  
**Pods** 229  
**Pods count** 164  
**policies**  
    circuit breakers 58  
    retries 57  
    timeout 57

**preview features, Dapr**  
    reference link 60

**Prod** 142

**Prometheus** 219, 227, 251  
    installing 219, 220  
    used, for analyzing metrics 219

**publish/subscribe (pub/sub)** 198, 206, 254

**pub/sub pattern**  
    implementation steps 103  
    using, in Dapr 99-102

**pub/sub pattern, implementation process**  
    Dapr component, configuring 106-108  
    messages, inspecting 109-111  
    topic, publishing 108, 109  
    topic, subscribing 104-106

**Python** 226  
    installation link 226

## R

**requests per second (RPS)** 238  
**resiliency** 57  
    applying 58-60  
**resiliency feature** 244  
**resource units, in Kubernetes**  
    reference link 233  
**retries** 57

## S

**saga pattern** 215  
    implementing 111-113  
    messages, publishing to Dapr 113  
    subscribing, to Dapr topic 114, 115  
    testing 115-117  
**sales-bounded context**  
    Reservation service 48

- sample solution** 39, 40
- scalable microservices** 49
- scaling** 228
- self-hosted gateway** 195
- self-hosted mode**
  - Dapr, installing 12, 13
- service** 41
- service discovery** 57
- service invocation, with .NET SDK** 60
  - Dapr, configuring in ASP.NET 61
  - Dapr, implementing with ASP.NET controller 61-63
  - Dapr, implementing with ASP.NET Minimal API 64-68
  - debugging configuration, preparing 64
  - project, creating for Order service 60, 61
  - project, creating for Reservation service 63
- service-level agreement (SLA)** 41
- service-oriented architecture (SOA)** 40
- services** 54
  - invoking, with Dapr 54, 55
- Services and ingresses** 165
- service-to-service invocation** 55-57
- Short Message Service (SMS)** 8
- Software as a Service (SaaS)** 250
- software development kit (SDK)** 6, 102, 145, 218
- span** 215
- state** 80
- state components**
  - reference link 81
- stateful reservation-service** 86
- stateful services, e-commerce**
  - ordering system 85
- Dapr state, handling in ASP.NET 86-89
- stateful reservation-service 86

- stateless microservices** 49
- state management, Dapr**
  - concurrency 82
  - consistency 82
  - reference link, for concurrency 82
  - reference link, for consistency 82
  - resiliency 84
  - stateful microservice 80
  - stateless microservice 80
  - state store 80, 81
  - state store interaction 83, 84
  - transactions 81, 82
- stock-keeping unit (SKU)** 116, 200, 215, 264
- sustainable software engineering** 43
  - reference link 43

## T

- timeout** 57
- tracing** 215
  - configuration, with Zipkin 210, 211
  - enabling, in Dapr 211-213
- Twilio output bindings**
  - configuring 124, 125
  - notification, verifying 127, 128
  - signaling via 125-127
  - using, in Dapr 122, 123
- Twilio trial**
  - signing up for 123, 124
  - URL 123
- Tye**
  - installing 35
  - using 35-38
  - using, with Dapr 35

## U

**unique identifier (UID)** 155

## V

**virtual actor pattern** 136-140

**virtual actors** 136

**Visual Studio Code, on Windows**

reference link 11

**Visual Studio Code (VS Code)** 9, 21, 214

Dapr extension 11

installing 11

reference link 11

**Visual Studio Code (VS Code) extension** 189

## W

**Windows Subsystem for Linux (WSL2)** 226

reference link 226

**Windows Terminal**

installing 11

reference link 11

**worker node** 234

## X

**XML** 198

## Y

**YAML** 251

**Yet Another Reverse Proxy (YARP)** 259

## Z

**Zipkin** 208, 227

investigating with 214-218

setting up 208-210

tracing with 208

URL 208

used, for tracing configuration 210, 211



Packt . com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

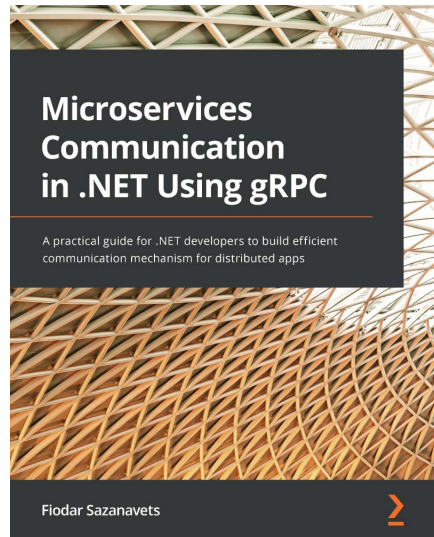
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [packt . com](http://packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub . com](mailto:customercare@packtpub.com) for more details.

At [www . packt . com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

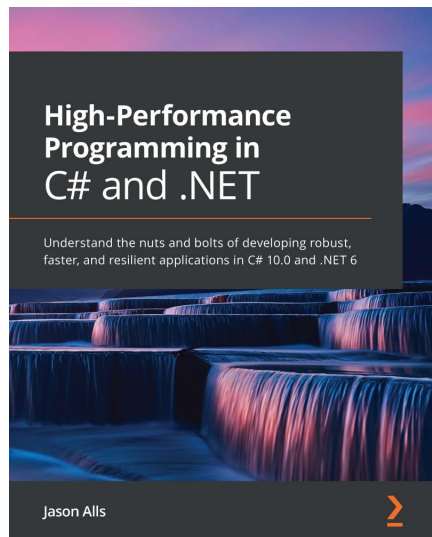


## **Microservices Communication in .NET Using gRPC**

Fiodar Sazanavets

ISBN: 9781803236438

- Get to grips with the fundamentals of gRPC and Protobuf
- Debug gRPC components inside a .NET application to locate and fix errors
- Understand gRPC best practices, such as performance enhancement
- Effectively translate between gRPC and native C# code by applying well-known types
- Secure gRPC communication inside a .NET application
- Discover how to monitor gRPC on .NET by applying logging and metrics



## High-Performance Programming in C# and .NET

Jason Alls

ISBN: 9781800564718

- Use correct types and collections to enhance application performance Profile, benchmark, and identify performance issues with the codebase
- Explore how to best perform queries on LINQ to improve an application's performance
- Effectively utilize a number of CPUs and cores through asynchronous programming
- Build responsive user interfaces with WinForms, WPF, MAUI, and WinUI
- Benchmark ADO.NET, Entity Framework Core, and Dapper for data access
- Implement CQRS and event sourcing and build and deploy microservices

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Share Your Thoughts

Now you've finished *Practical Microservices with Dapr and .NET - Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, [please click here to go straight to the Amazon review page for this book](#) and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

---

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/978-1-80324-812-7>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly