

Dapr for .NET Developers

Foreword by Mark Russinovich, Microsoft Azure CTO
and Technical Fellow



Robert Vettor
Sander Molenkamp
Edwin van Wijk

PREVIEW EDITION

PUBLISHED BY

Microsoft Developer Division, .NET, and Azure Incubations teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2021 by Microsoft Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

This book is provided “as-is” and expresses the author’s views and opinions. The views, opinions, and information expressed in this book, including URL and other Internet website references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

Microsoft and the trademarks listed at <https://www.microsoft.com> on the “Trademarks” webpage are trademarks of the Microsoft group of companies.

Mac and macOS are trademarks of Apple Inc.

The Docker whale logo is a registered trademark of Docker, Inc. Used by permission.

All other marks and logos are property of their respective owners.

Authors:

Rob Vettor, Principal Cloud Solution Architect - thinkingincloudnative.com, Microsoft

Sander Molenkamp, Principal Cloud Architect/Microsoft MVP - sander.molenkamp.com, [Info Support](#)

Edwin van Wijk, Principal Solution Architect/Microsoft MVP - defaultconstructor.com, [Info Support](#)

Participants and Reviewers:

Mark Russinovich, Azure CTO and Technical Fellow, Azure Office of CTO, Microsoft

Nish Anil, Senior Program Manager, .NET team, Microsoft

Mark Fussell, Principal Program Manager, Azure Incubations, Microsoft

Yaron Schneider, Principal Software Engineer, Azure Incubations, Microsoft

Ori Zohar, Senior Program Manager, Azure Incubations, Microsoft

Editors:

David Pine, Senior Content Developer, .NET team, Microsoft

Maira Wenzel, Senior Program Manager, .NET team, Microsoft

Version

This guide has been written to cover the **Dapr 1.0** version. .NET Core samples are based on **.NET Core 3.1**.

Who should use this guide

The audience for this guide is mainly developers, development leads, and architects who are interested in learning how to build applications designed for the cloud.

A secondary audience is technical decision-makers who plan to choose whether to build their applications using a cloud-native approach.

How you can use this guide

This guide is available both in [PDF](#) form and online. Feel free to forward this document or links to its online version to your team to help ensure common understanding of these topics. Most of these topics benefit from a consistent understanding of the underlying principles and patterns, as well as the trade-offs involved in decisions related to these topics. Our goal with this document is to equip teams and their leaders with the information they need to make well-informed decisions for their applications' architecture, development, and hosting.

Send your feedback

This book and related samples are constantly evolving, so your feedback is welcomed! If you have comments about how this book can be improved, use the feedback section at the bottom of any page built on [GitHub issues](#).

Contents

Foreword	1
The world is distributed.....	3
Summary	7
Dapr at 20,000 feet	8
Dapr and the problem it solves?.....	8
Dapr architecture	9
Building blocks	9
Components.....	11
Sidecar architecture.....	14
Hosting environments.....	14
Dapr performance considerations	15
Dapr and service meshes.....	16
Summary	18
References.....	18
Get started with Dapr	19
Install Dapr into your local environment.....	19
Build your first Dapr application.....	19
Create the application	19
Add Dapr State Management	20
Component configuration files	21
Build a multi-container Dapr application.....	23
Create the application	24
Add Dapr service invocation.....	27
Add container support.....	29
Summary	33
References.....	34
Dapr reference application.....	35
eShop on containers.....	35

eShop on Dapr	36
Application of Dapr building blocks.....	38
Benefits of applying Dapr to eShop	38
Summary	39
References.....	40
The Dapr state management building block.....	41
What it solves	41
How it works	42
Consistency.....	43
Concurrency	45
Transactions	45
Use the Dapr .NET SDK.....	46
ASP.NET Core integration	47
State store components.....	48
Configuration	48
Key prefix strategies.....	49
Reference application: eShopOnDapr	50
Summary	52
References.....	52
The Dapr service invocation building block	53
What it solves	53
How it works	53
Use the Dapr .NET SDK.....	55
Invoke HTTP services using HttpClient.....	55
Invoke HTTP services using DaprClient	57
Invoke gRPC services using DaprClient.....	58
Reference application: eShopOnDapr	58
Forward HTTP requests using Envoy and Dapr	60
Make aggregated service calls using the .NET SDK	61
Summary	64
References.....	64

The Dapr publish & subscribe building block	65
What it solves	65
How it works	66
Competing consumers	70
SDKs	70
Use the Dapr .NET SDK	70
Pub/sub components	72
Configure pub/sub components	72
Reference application: eShopOnDapr	74
Publish events	74
Subscribe to events	75
Use pub/sub components	76
Summary	77
The Dapr bindings building block	78
What it solves	78
How it works	79
Input bindings	79
Output bindings	80
Using the Dapr .NET SDK	82
Binding components	82
Cron binding	83
Reference application: eShopOnDapr	84
Summary	85
References	85
The Dapr observability building block	86
What it solves	87
How it works	87
Distributed tracing	88
Metrics	94
Logging	97
Health status	99

Dapr dashboard	100
Use the Dapr .NET SDK	102
Reference application: eShopOnDapr	102
Custom health dashboard	102
Seq log aggregator	103
Application Insights.....	103
Summary	103
References	104
The Dapr secrets building block	105
What it solves	105
How it works	106
Use the Dapr .NET SDK	107
Secret store components.....	108
Configuration.....	109
Indirectly consume Dapr secrets.....	109
Local file	110
Kubernetes secret	112
Azure Key Vault.....	112
Scope secrets	115
Reference application: eShopOnDapr	115
Summary	117
References	117
Summary and the road ahead.....	118
The road ahead.....	121

Foreword

With the wave of cloud adoption well underway, there is a major shift happening towards “cloud native” development, often built with microservice-architectures. These microservices are both stateless and stateful, and run on the cloud and edge, embracing the diversity of languages and frameworks available today. This enterprise shift is driven by both the market forces of faster time to market, as well as the scale and efficiencies of building services for the cloud. Even before COVID-19, cloud adoption was accelerating for enterprises and developers were being asked to do even more to deliver on building these distributed system applications, and that has only accelerated since. Developers in enterprises seek to focus on business logic, while leaning on platforms to imbue their applications with scale, resiliency, maintainability, elasticity, and the other attributes of cloud-native architectures, which is why there is also shift towards serverless platforms that hide the underlying infrastructure. Developers should not be expected to become distributed systems experts. This is where Dapr steps in to help you, whether you are building on infrastructure such as Kubernetes, or on a serverless platform.

Dapr is designed as an enterprise, developer-focused, microservices programming model platform with the mantra “any language, any framework, run anywhere”. It makes building distributed applications easy and portable across any infrastructure, from public-cloud, through hierarchical edge, and even down to single node IoT devices. It emerged from both our experiences building services in Azure as well as time spent working with customers building applications on Azure Kubernetes Service and Azure Service Fabric. Over and over, we saw common problems that they had to address. It became clear that there was a need to provide a “library” of common microservice best practices that developers could use, not only in new greenfield applications, but also to aid in the modernization of existing applications. In the containerized, distributed, and networked cloud native world, the sidecar model has emerged as the preferred approach, in the same way DLLs are preferred in the client/server generation. Using Dapr’s sidecar and APIs give you, as a developer, all the power of distributed systems functionality, with the ease of a single HTTP or gRPC local call.

To address the wide range of scenarios that developers face, Dapr provides features such as state management, service to service invocation, pub/sub and integration to external systems with I/O bindings, which are based on the triggers and bindings of Azure Functions. These in turn take advantage of Dapr’s component model which allows you to “swap out”, say different underlying state stores, without having to change any code, making code more portable, more flexible and allowing for experimentation of what best suits your needs. Developers don’t need to learn and incorporate service SDKs into their code, worry about authentication, secret management, retries or conditional code that targets specific deployment environments.

This book shows how Dapr reduces your development time and overall code maintenance by incrementally “Daperizing” the canonical .NET reference application, eShop. For example, in the

original eShop implementation, significant amounts of code were written to abstract between Azure Service Bus and RabbitMQ for publishing events between services. All this code can be discarded and simply replaced with Dapr's pub/sub API and component model which had an even wider range of pub/sub brokers, rather than just two. Dapr's actor model, when used in the reworked eShop application, shows the ease of building long running, stateful, event driven, workflow applications with all the difficulties of concurrency and multi-threading removed. By the end of this book, you will see the drastic simplification that Dapr brings to your application development, and I firmly believe all developers embarking on a cloud native app building journey should leverage Dapr.

We publicly announced Dapr with the v0.1 release in Oct 2019 and now, a year and half later, I am thrilled to say that Dapr is ready for production usage with the v1.0 release. Getting Dapr to v1.0 has truly been a community effort. It has been amazing to see the open-source community coalesce around Dapr and grow since it was first announced – from 114 contributors in October 2019 to over 700 in early 2021 - a six-fold increase in 16 months! Contributions to the project have gone to every Dapr repo and have ranged from opening issues, commenting on feature proposals, providing samples, and of course contributing code. The parts of the project community members have contributed to the most include the Dapr runtime, docs, CLI, SDKs and the creation of a rich ecosystem of components. Maintaining this openness is critical to Dapr's future.

Dapr is really just getting started, though, and you should expect to see more Dapr capabilities and more support for Dapr in Azure services. I hope that you will take advantage of Dapr to enable you to focus on your core business logic and accelerate your microservices development. I am excited to have you join us in the Dapr community on this journey at <https://github.com/dapr/> and on Discord <https://aka.ms/dapr-discord>.

Modern distributed systems are complex. You start with small, loosely coupled, independently deployable services. These services cross process and server boundaries. They then consume different kinds of infrastructure backing services (databases, message brokers, key vaults). Finally, these disparate pieces compose together to form an application.

Mark Russinovich Azure CTO and Technical Fellow Microsoft

The world is distributed

Just ask any 'cool kid': *Modern, distributed systems are in, and monolithic apps are out!*

But, it's not just "cool kids." Progressive IT Leaders, corporate architects, and astute developers are echoing these same thoughts as they explore and evaluate modern distributed applications. Many have bought in. They're designing new and replatforming existing enterprise applications following the principles, patterns, and practices of distributed microservice applications.

But, this evolution raises many questions...

- What exactly is a distributed application?
- Why are they gaining popularity?
- What are the costs?
- And, importantly, what are the tradeoffs?

To start, let's rewind and look at the past 15 years. During this period, we typically constructed applications as a single, monolithic unit. Figure 1-1 shows the architecture.

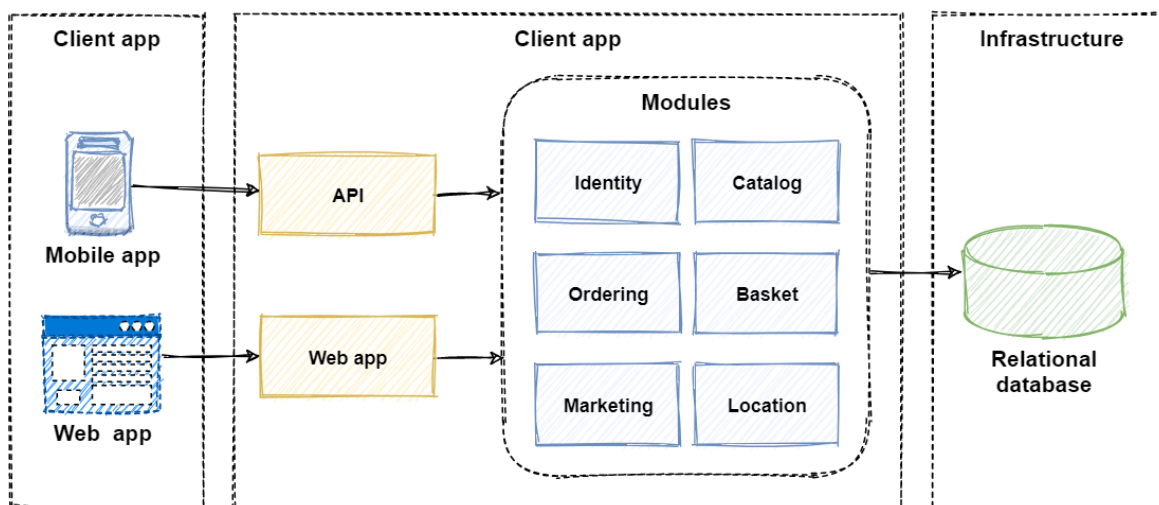


Figure 1-1. Monolithic architecture.

Note how the modules for Ordering, Identity, and Marketing execute in a single-server process. Application data is stored in a shared database. Business functionality is exposed via HTML and RESTful interfaces.

In many ways, monolithic apps are straightforward. They're straightforward to:

- Build

- Test
- Deploy
- Troubleshoot
- Scale vertically (scale up)

However, monolithic architectures can present significant challenges.

Over time, you may reach a point where you begin to lose control...

- The monolith has become so overwhelmingly complicated that no single person understands it.
- You fear making changes as each brings unintended and costly side effects.
- New features/fixes become time-consuming and expensive to implement.
- Even the smallest change requires full deployment of the entire application - expensive and risky.
- One unstable component can crash the entire system.
- Adding new technologies and frameworks aren't an option.
- Implementing agile delivery methodologies are difficult.
- Architectural erosion sets in as the code base deteriorates with never-ending "special cases."
- Eventually the consultants come in and tell you to rewrite it.

IT practitioners call this condition the Fear Cycle. If you've been in the technology business for any length of time, good chance you've experienced it. It's stressful and exhausts your IT budget. Instead of building new and innovative solutions, most of your budget is spent maintaining legacy apps.

Instead of fear, businesses require speed and agility. They seek an architectural style with which they can rapidly respond to market conditions. They need to instantaneously update and individually scale small areas of a live application.

An early attempt to gain speed and agility came in the form of [Service Oriented Architecture](#), or SOA. In this model, service consumers and service providers collaborated via middleware messaging components, often referred to as an [Enterprise Service Bus](#), or ESB. Figure 1-2 shows the architecture.

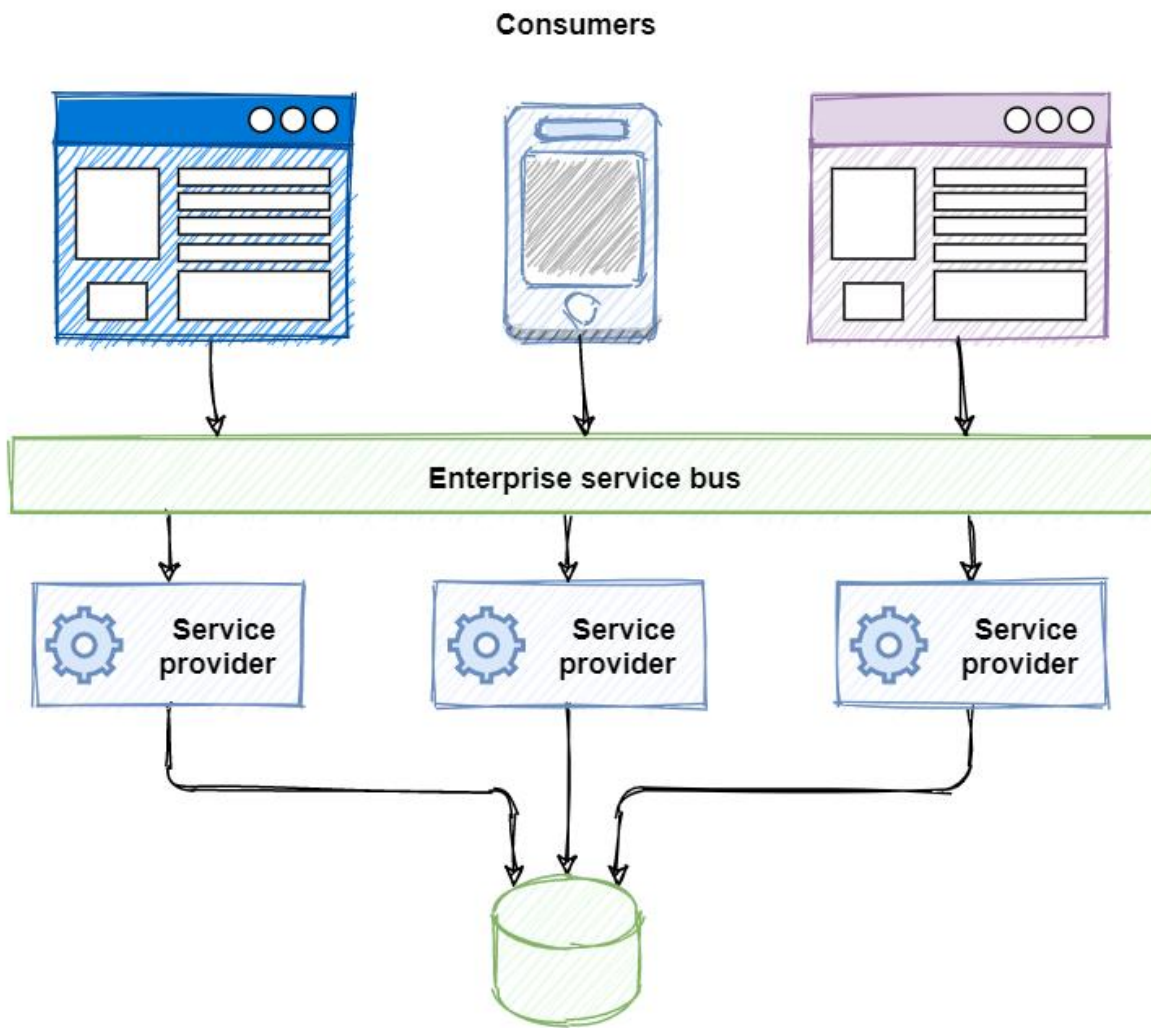


Figure 1-2. SOA architecture.

With SOA, centralized service providers registered with the ESB. Business logic would be built into the ESB to integrate providers and consumers. Service consumers could then find and communicate with these providers using the ESB.

Despite the promises of SOA, implementing this approach often increased complexity and introduced bottlenecks. Maintenance costs became high and ESB middleware expensive. Services tended to be large. They often shared dependencies and data storage. In the end, SOAs often resulted in a 'distributed monolithic' structure with centralized services that were resistant to change.

Nowadays, many organizations have realized speed and agility by adopting a distributed microservice architectural approach to building systems. Figure 1-3 shows the same system built using distributed techniques and practices.

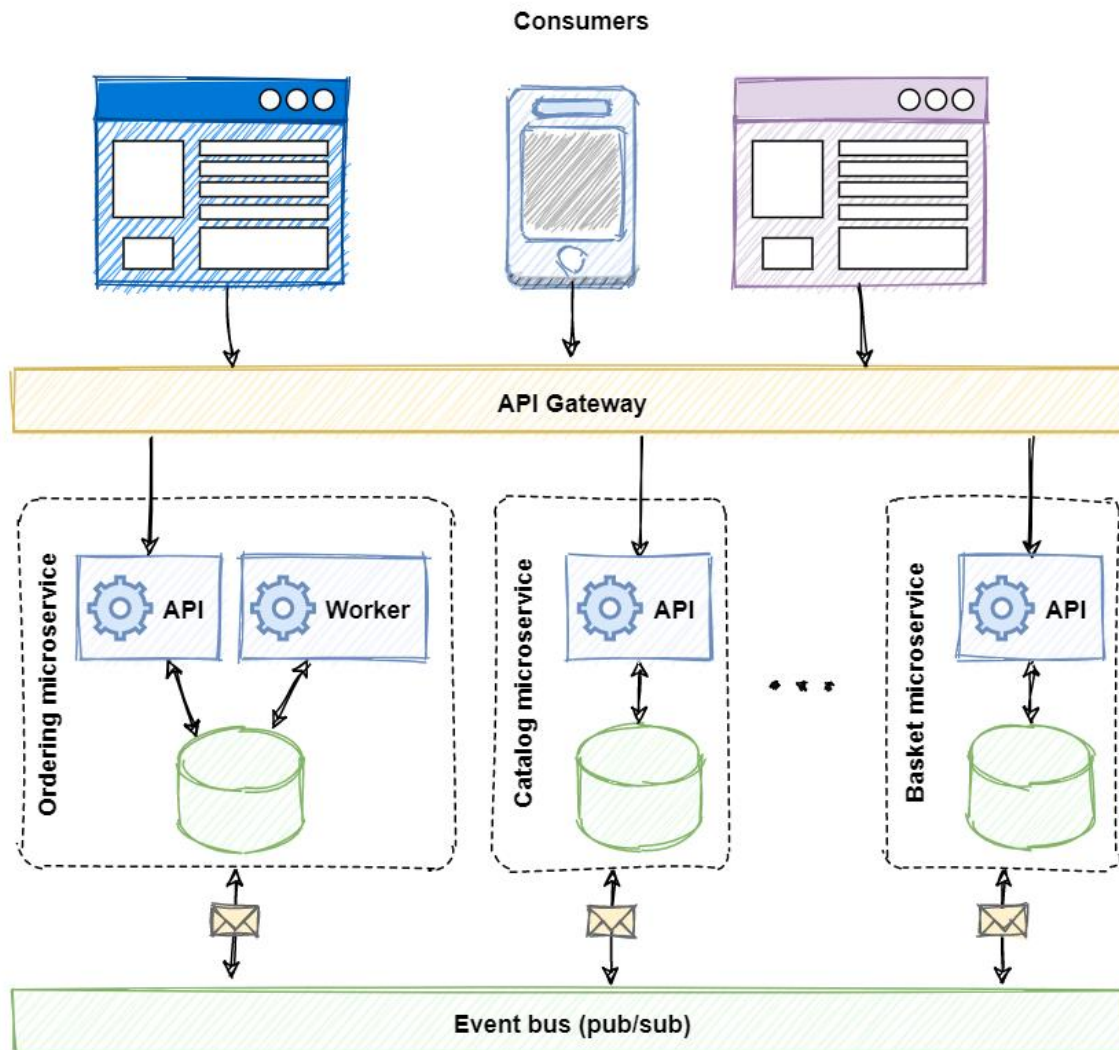


Figure 1-3. Distributed architecture.

Note how the same application is decomposed across a set of distributed services. Each is self-contained and encapsulates its own code, data, and dependencies. Each is deployed in a software container and managed by a container orchestrator. Instead of a single database shared by multiple services, each service owns a private database. Other services can't access this database directly and can only get to data that is exposed through the public API of the service that owns it. Note how some services require a full relational database, but others, a NoSQL datastore. The basket service stores its state in a distributed key/value cache. Note how inbound traffic routes through an API Gateway service. It's responsible for directing calls to services and enforcing cross-cutting concerns. Most importantly, the application takes full advantage of the scalability, availability, and resiliency features found in modern cloud platforms.

But, while distributed services can provide agility and speed, they present a different set of challenges. Consider the following...

- How can distributed services discover each other and communicate synchronously?

- How can they implement asynchronous messaging?
- How can they maintain contextual information across a transaction?
- How can they become resilient to failure?
- How can they scale to meet fluctuating demand?
- How are they monitored and observed?

For each of these challenges, multiple products are often available. But, shielding your application from product differences and keeping code maintainable and portable become a challenge.

This book introduces Dapr. Dapr is a distributed application runtime. It directly addresses many of the challenges found that come along with distributed applications. Looking ahead, Dapr has the potential to have a profound impact on distributed application development.

Summary

In this chapter, we discussed the adoption of distributed applications. We contrasted a monolithic system approach with that of distributed services. We pointed out many of the common challenges when considering a distributed approach.

Now, sit back, relax, and let us introduce you the new world of Dapr.

Dapr at 20,000 feet

In chapter 1, we discussed the appeal of distributed microservice applications. But, we also pointed out that they dramatically increase architectural and operational complexity. With that in mind, the question becomes, how can you “have your cake” and “eat it too?”. That is, how can you take advantage of the agility of distributed architecture, and minimize its complexity?

Dapr, or *Distributed Application Runtime*, is a new way to build modern distributed applications.

What started as a prototype has evolved into a highly successful open-source project. Its sponsor, Microsoft, has closely partnered with customers and the open-source community to design and build Dapr. The Dapr project brings together developers from all backgrounds to solve some of the toughest challenges of developing distributed applications.

This book looks at Dapr from the viewpoint of a .NET developer. In this chapter, you’ll build a conceptual understanding of Dapr and how it works. Later on, we present practical, hands-on instruction on how you can use Dapr in your applications.

Imagine flying in a jet at 20,000 feet. You look out the window and see the landscape below from a wide perspective. Let’s do the same for Dapr. Visualize yourself flying over Dapr at 20,000 feet. What would you see?

Dapr and the problem it solves?

Dapr addresses a large challenge inherent in modern distributed applications: **Complexity**.

Through an architecture of pluggable components, Dapr greatly simplifies the plumbing behind distributed applications. It provides a **dynamic glue** that binds your application with infrastructure capabilities from the Dapr runtime. For example, your application may require a state store. You could write custom code to target Redis Cache and inject it into your service at runtime. However, Dapr simplifies your experience by providing a distributed cache capability out-of-the-box. Your service invokes a Dapr **building block** that dynamically binds to Redis Cache **component** via a Dapr **configuration**. With this model, your service delegates the call to Dapr, which calls Redis on your behalf. Your service has no SDK, library, or direct reference to Redis. You code against the common Dapr state management API, not the Redis Cache API.

Figure 2-1 shows Dapr from 20,000 feet.

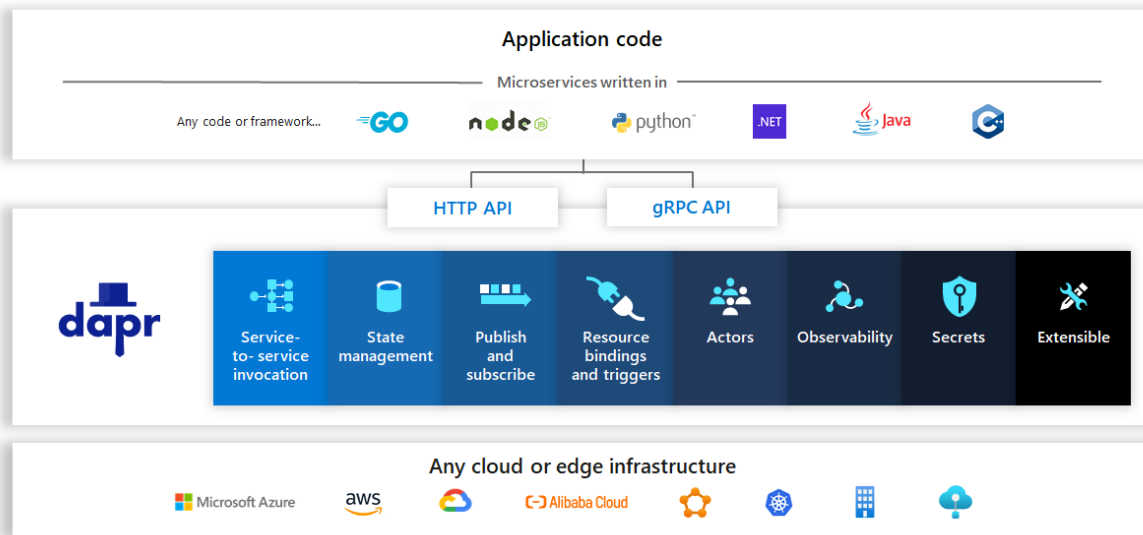


Figure 2-1. Dapr at 20,000 feet.

In the top row of the figure, note how Dapr provides language-specific SDKs for popular development platforms. Dapr v 1.0 includes supports Go, Node.js, Python, .NET, Java, and JavaScript. This book focuses on the Dapr .NET SDK, which also provides direct support for ASP.NET Core integration.

While language-specific SDKs enhance the developer experience, Dapr is platform agnostic. Under the hood, Dapr’s programming model exposes capabilities through standard HTTP/gRPC communication protocols. Any programming platform can call Dapr via its native HTTP and gRPC APIs.

The blue boxes across the center of the figure represent the Dapr building blocks. Each exposes a distributed application capability that your application can consume.

The bottom row highlights the portability of Dapr and the diverse environments across which it can run.

Dapr architecture

At this point, the jet turns around and flies back over Dapr, descending in altitude, giving you a closer look at how Dapr works.

Building blocks

From the new perspective, you see a more detailed view of the Dapr **building blocks**.

A building block encapsulates a distributed infrastructure capability. You can access the functionality through the HTTP or gRPC APIs. Figure 2-2 shows the available blocks for Dapr v 1.0.

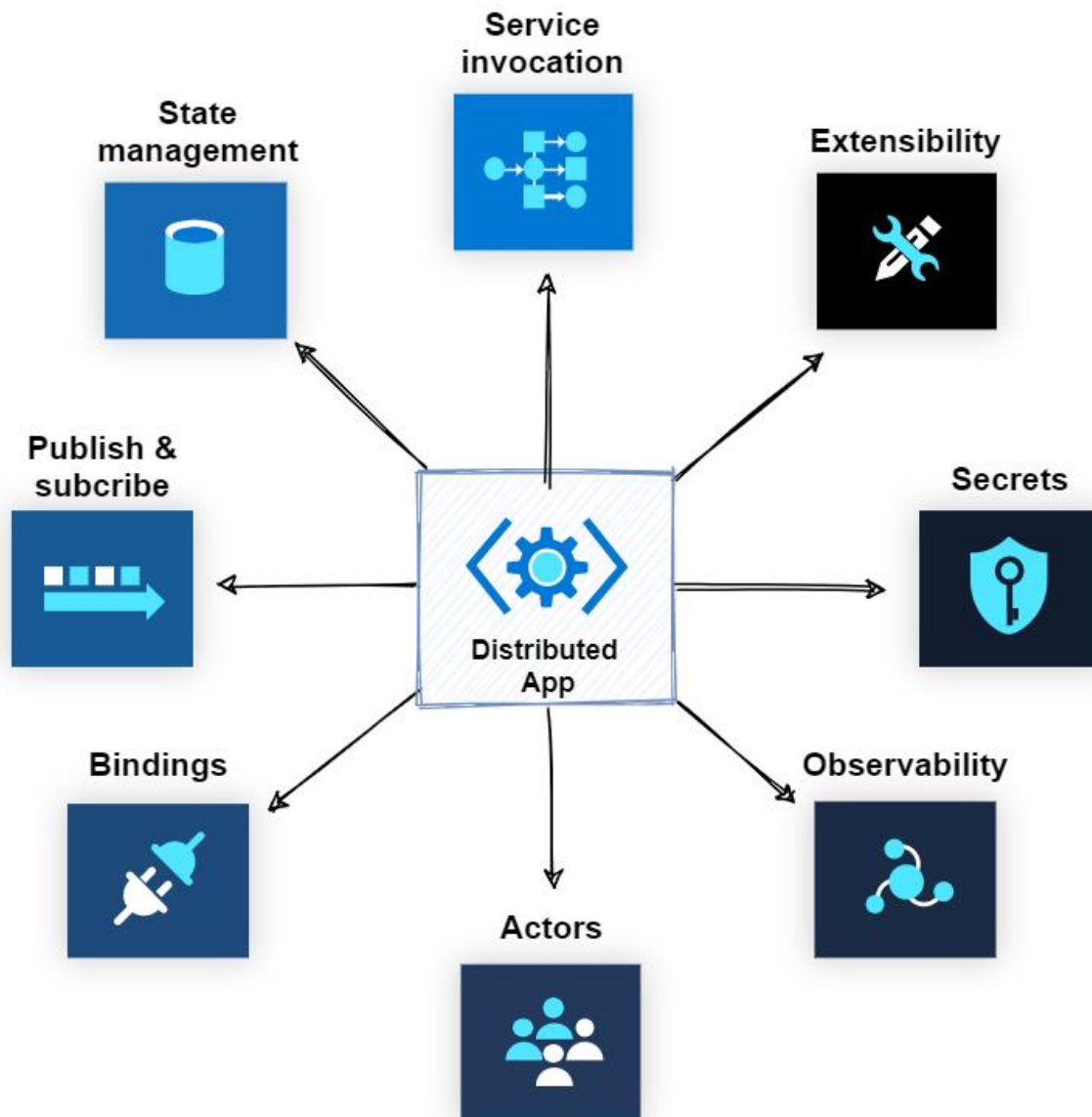


Figure 2-2. Dapr building blocks.

The following table describes the infrastructure services provided by each block.

Building block	Description
State management	Support contextual information for long running stateful services.
Service invocation	Invoke direct, secure service-to-service calls using platform agnostic protocols and well-known endpoints.
Publish and subscribe	Implement secure, scalable pub/sub messaging between services.

Building block	Description
Bindings	Trigger code from events raised by external resources with bi-directional communication.
Observability	Monitor and measure message calls across networked services.
Secrets	Securely access external secret stores.
Actors	Encapsulate logic and data in reusable actor objects.

Building blocks abstract the implementation of distributed application capabilities from your services. Figure 2-3 shows this interaction.

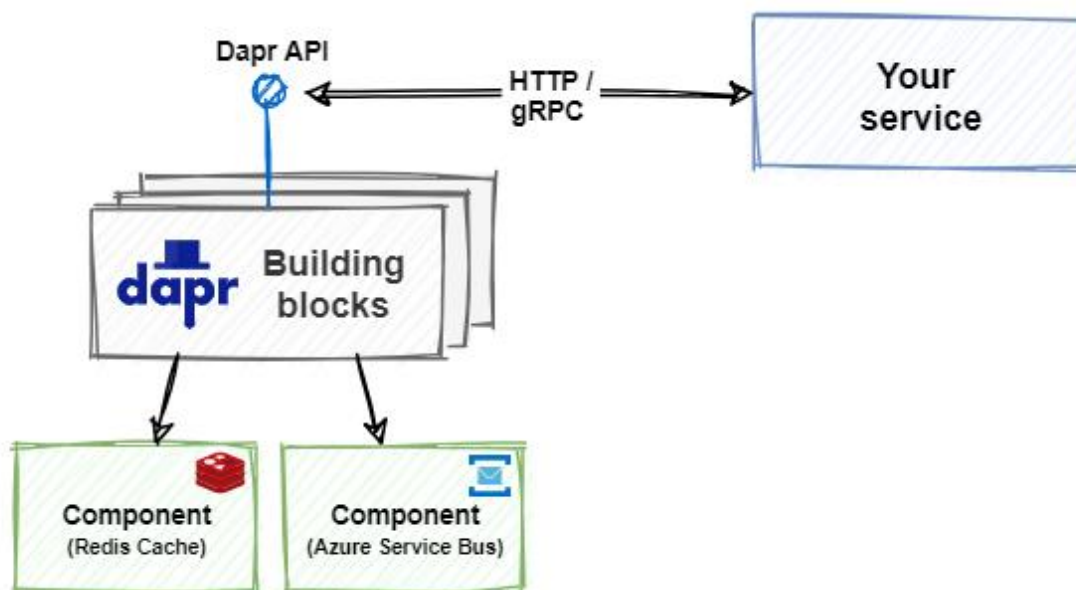


Figure 2-3. Dapr building block integration.

Building blocks invoke Dapr components that provide the concrete implementation for each resource. The code for your service is only aware of the building block. It takes no dependencies on external SDKs or libraries - Dapr handles the plumbing for you. Each building block is independent. You can use one, some, or all of them in your application. As a value-add, Dapr building blocks bake in industry best practices including comprehensive observability.

We provide detailed explanation and code samples for each Dapr building block in the upcoming chapters. At this point, the jet descends even more. From the new perspective, you now have a closer look at the Dapr components layer.

Components

While building blocks expose an API to invoke distributed application capabilities, Dapr components provide the concrete implementation to make it happen.

Consider, the Dapr **state store** component. It provides a uniform way to manage state for CRUD operations. Without any change to your service code, you could switch between any of the following Dapr state components:

- AWS DynamoDB
- Aerospike
- Azure Blob Storage
- Azure CosmosDB
- Azure Table Storage
- Cassandra
- Cloud Firestore (Datastore mode)
- CloudState
- Couchbase
- Etcd
- HashiCorp Consul
- Hazelcast
- Memcached
- MongoDB
- PostgreSQL
- Redis
- RethinkDB
- SQL Server
- Zookeeper

Each component provides the necessary implementation through a common state management interface:

```
type Store interface {  
    Init(metadata Metadata) error  
    Delete(req *DeleteRequest) error  
    BulkDelete(req []DeleteRequest) error  
    Get(req *GetRequest) (*GetResponse, error)  
    Set(req *SetRequest) error  
    BulkSet(req []SetRequest) error  
}
```

Tip

The Dapr runtime as well as all of the Dapr components have been written in the Golang, or Go, language. Go is a popular language across the open source community and attests to cross-platform commitment of Dapr.

Perhaps you start with Azure Redis Cache as your state store. You specify it with the following configuration:

```

apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: statestore
  namespace: default
spec:
  type: state.redis
  version: v1
  metadata:
    - name: redisHost
      value: <HOST>
    - name: redisPassword
      value: <PASSWORD>
    - name: enableTLS
      value: <bool> # Optional. Allowed: true, false.
    - name: failover
      value: <bool> # Optional. Allowed: true, false.

```

In the **spec** section, you configure Dapr to use the Redis Cache for state management. The section also contains component-specific metadata. In this case, you can use it to configure additional Redis settings.

At a later time, the application is ready to go to production. For the production environment, you may want to change your state management to Azure Table Storage. Azure Table Storage provides state management capabilities that are affordable and highly durable.

At the time of this writing, the following component types are provided by Dapr:

Component	Description
Service discovery	Used by the service invocation building block to integrate with the hosting environment to provide service-to-service discovery.
State	Provides a uniform interface to interact with a wide variety of state store implementations.
Pub/sub	Provides a uniform interface to interact with a wide variety of message bus implementations.
Bindings	Provides a uniform interface to trigger application events from external systems and invoke external systems with optional data payloads.
Middleware	Allows custom middleware to plug into the request processing pipeline and invoke additional actions on a request or response.
Secret stores	Provides a uniform interface to interact with external secret stores, including cloud, edge, commercial, open-source services.
Tracing exporters	Provides a uniform interface to open telemetry wrappers.

As the jet completes its fly over of Dapr, you look back once more and can see how it connects together.

Sidecar architecture

Dapr exposes its building blocks and components through a [sidecar architecture](#). A sidecar enables Dapr to run in a separate memory process or separate container alongside your service. Sidecars provide isolation and encapsulation as they aren't part of the service, but connected to it. This separation enables each to have its own runtime environment and be built upon different programming platforms. Figure 2-4 shows a sidecar pattern.

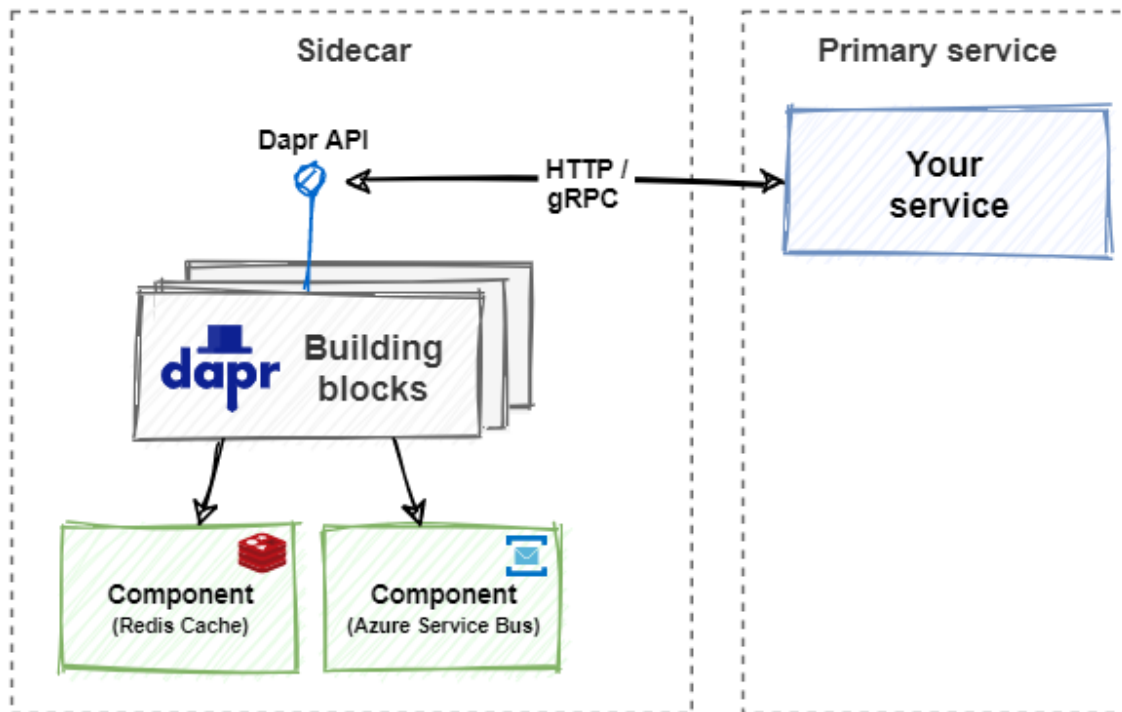


Figure 2-4. Sidecar architecture.

This pattern is named Sidecar because it resembles a sidecar attached to a motorcycle. In the previous figure, note how the Dapr sidecar is attached to your service to provide distributed application capabilities.

Hosting environments

Dapr has cross-platform support and can run in many different environments. These environments include Kubernetes, a group of VMs, or edge environments such as Azure IoT Edge.

For local development, the easiest way to get started is with [self-hosted mode](#). In self-hosted mode, the microservices and Dapr sidecars run in separate local processes without a container orchestrator such as Kubernetes. For more information, see [download and install the Dapr CLI](#).

Figure 2-5 shows an application and Dapr hosted in two separate memory processes communicating via HTTP or gRPC.

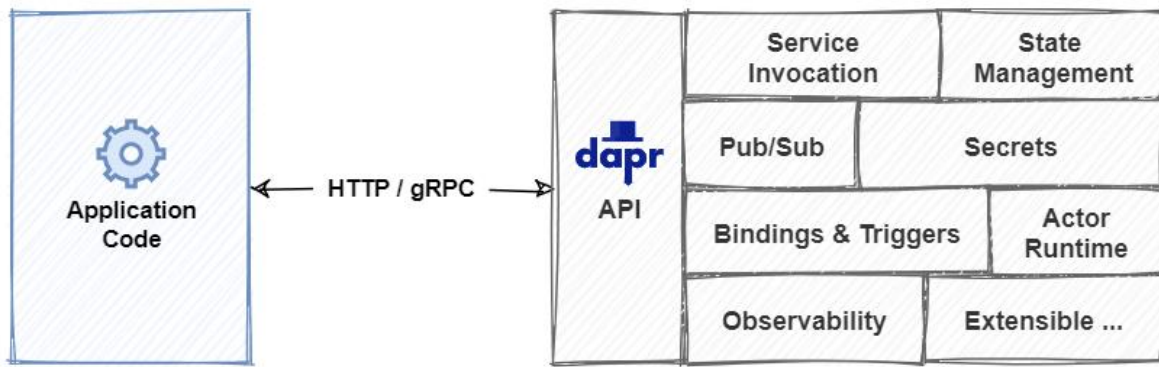


Figure 2-5. Self-hosted Dapr sidecar.

By default, Dapr installs Docker containers for Redis and Zipkin to provide default state management and observability. If you don't want to install Docker on your local machine, you can even [run Dapr in self-hosted mode without any Docker containers](#). However, you must install default components such as Redis for state management and pub/sub manually.

Dapr also runs in [containerized environments](#), such as Kubernetes. Figure 2-6 shows Dapr running in a separate side-car container along with the application container in the same Kubernetes pod.

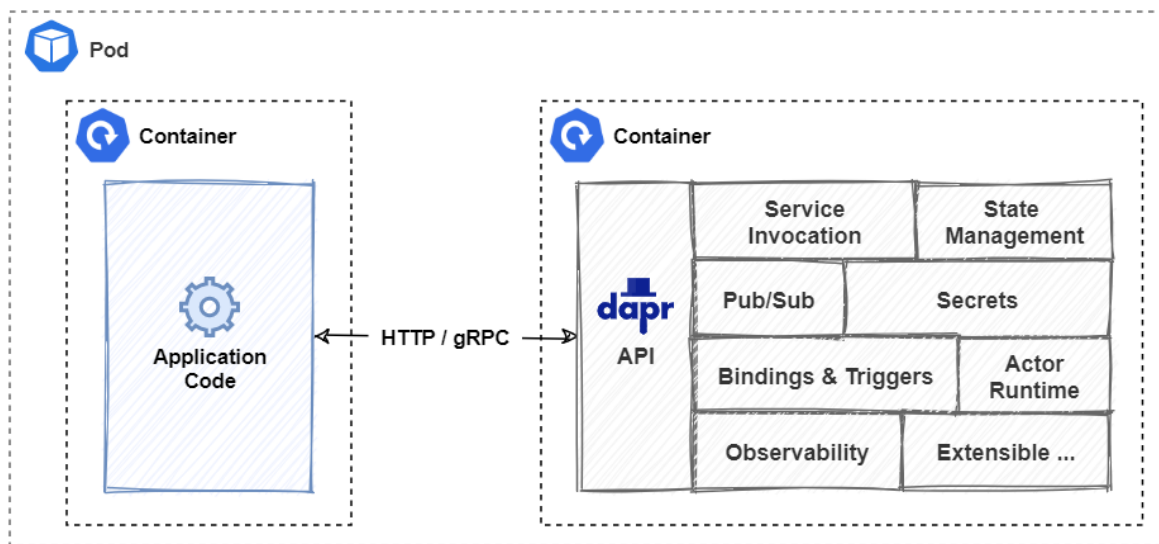


Figure 2-6. Kubernetes-hosted Dapr sidecar.

Dapr performance considerations

As you've seen, Dapr exposes a sidecar architecture to decouple your application from distributed application capabilities. Invoking a Dapr operation requires at least one out-of-process network call. Figure 2-7 presents an example of a Dapr traffic pattern.

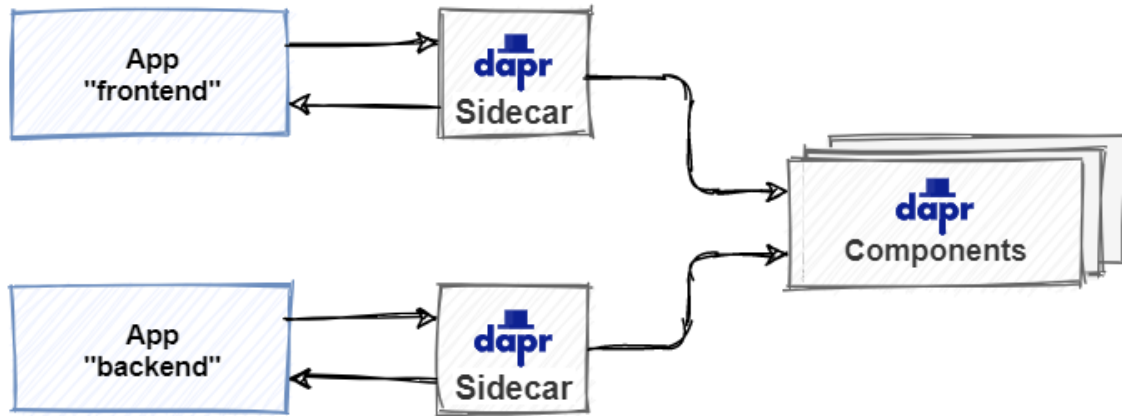


Figure 2-7. Dapr traffic patterns.

Looking at the previous figure, one might question the latency and overhead incurred for each call.

The Dapr team has invested heavily in performance. A tremendous amount of engineering effort has gone into making Dapr efficient. Calls between Dapr sidecars are always made with gRPC, which delivers high performance and small binary payloads. In most cases, the additional overhead should be sub-millisecond.

To increase performance, developers can call the Dapr building blocks with gRPC.

gRPC is a modern, high-performance framework that evolves the age-old [remote procedure call \(RPC\)](#) protocol. gRPC uses HTTP/2 for its transport protocol, which provides significant performance enhancements over HTTP RESTful service, including:

- Multiplexing support for sending multiple parallel requests over the same connection - HTTP 1.1 limits processing to one request/response message at a time.
- Bidirectional full-duplex communication for sending both client requests and server responses simultaneously.
- Built-in streaming enabling requests and responses to asynchronously stream large data sets.

Dapr and service meshes

Service mesh is another rapidly evolving technology for distributed applications.

A service mesh is a configurable infrastructure layer with built-in capabilities to handle service-to-service communication, resiliency, load balancing, and telemetry capture. It moves the responsibility for these concerns out of the services and into the service mesh layer. Like Dapr, a service mesh also follows a sidecar architecture.

Figure 2-8 shows an application that implements service mesh technology.

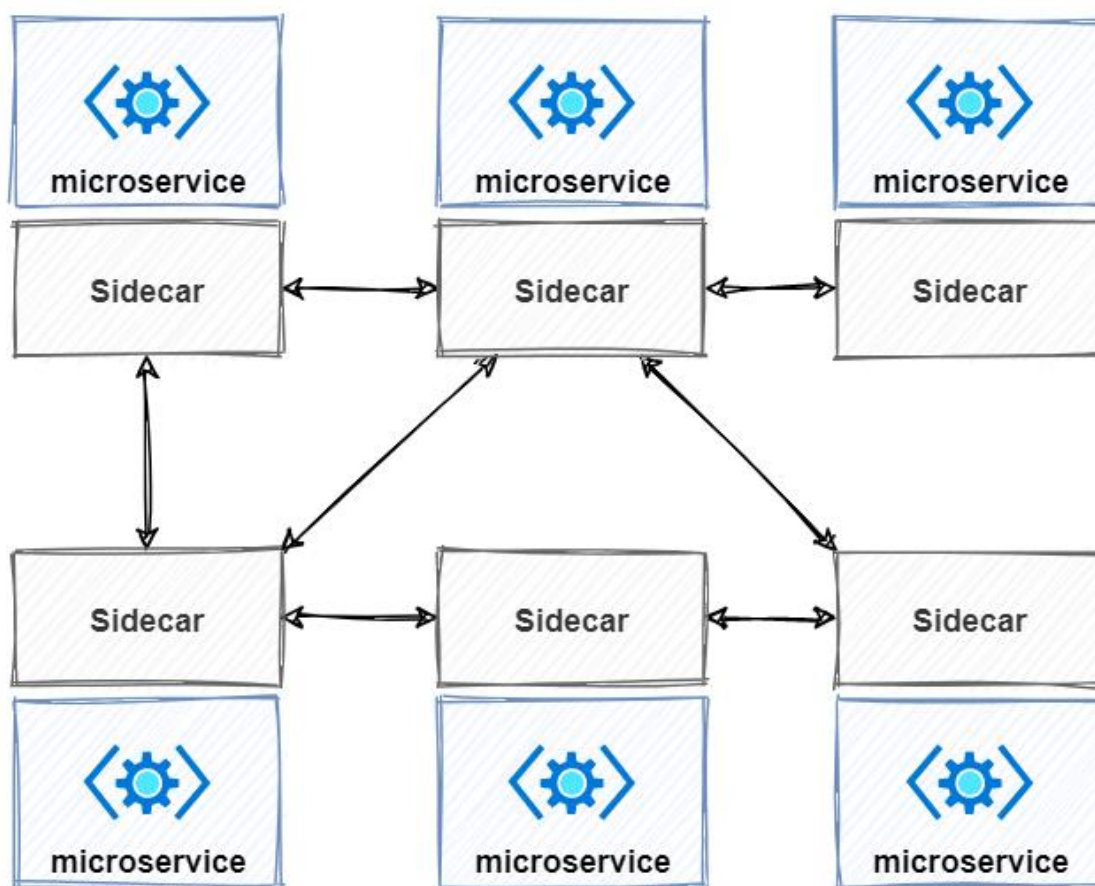


Figure 2-8. Service mesh with a side car.

The previous figure shows how messages are intercepted by a proxy that runs alongside each service. Each proxy can be configured with traffic rules specific to the service. It understands messages and can route them across your services and the outside world.

So the question becomes, "Is Dapr a service mesh?".

While both use a sidecar architecture, each technology has a different purpose. Dapr provides distributed application features. A service mesh provides a dedicated network infrastructure layer.

As each works at a different level, both can work together in the same application. For example, a service mesh could provide networking communication between services. Dapr could provide application services such as state management or actor services.

Figure 2-9 shows an application that implements both Dapr and service mesh technology.

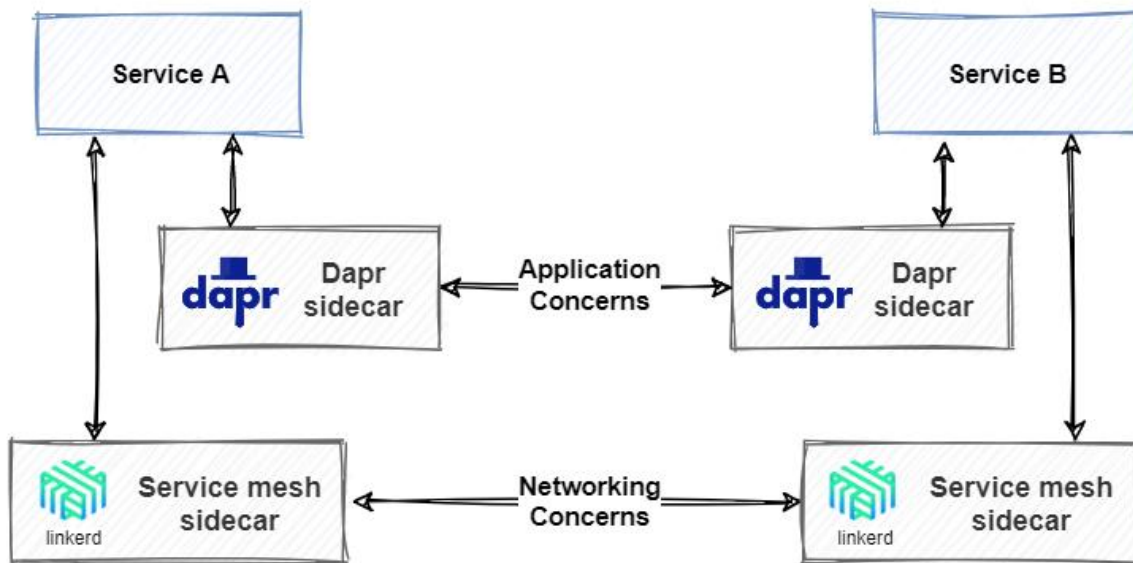


Figure 2-9. Dapr and service mesh together.

In the book, [Learning Dapr](#), authors Haishi Bai and Yaron Schneider, cover the integration of Dapr and service mesh.

Summary

This chapter introduced you to Dapr, a Distributed Application Runtime.

Dapr is an open-source project sponsored by Microsoft with close collaboration from customers and the open-source community.

At its core, Dapr helps reduce the inherent complexity of distributed microservice applications. It's built upon a concept of building block APIs. Dapr building blocks expose common distributed application capabilities, such as state management, service-to-service invocation, and pub/sub messaging. Dapr components lie beneath the building blocks and provide the concrete implementation for each capability. Applications bind to various components through configuration files.

In the next chapters, we present practical, hands-on instruction on how to use Dapr in your applications.

References

- [Dapr documentation](#)
- [Learning Dapr](#)
- [.NET Microservices: Architecture for Containerized .NET applications](#)
- [Architecting Cloud-Native .NET Apps for Azure](#)

Get started with Dapr

In the first two chapters, you learned basic concepts about Dapr. It's time to take it for a *test drive*. This chapter will guide you through preparing your local development environment and building two Dapr .NET applications.

Install Dapr into your local environment

You'll start by installing Dapr on your development computer. Once complete, you can build and run Dapr applications in [self-hosted mode](#).

1. [Install the Dapr CLI](#). It enables you to launch, run, and manage Dapr instances. It also provides debugging support.
2. Install [Docker Desktop](#). If you're running on Windows, make sure that **Docker Desktop for Windows** is configured to use Linux containers.

Note

By default, Dapr uses Docker containers to provide you the best out-of-the-box experience. To run Dapr outside of Docker, you can skip this step and [execute a slim initialization](#). The examples in this chapter require you use Docker containers.

1. [Initialize Dapr](#). This step sets up your development environment by installing the latest Dapr binaries and container images.
2. Install the [.NET Core 3 Development Tools](#) for .NET Core 3.1.

Now that Dapr is installed, it's time to build your first Dapr application!

Build your first Dapr application

You'll start by building a simple .NET Console application that consumes the [Dapr state management](#) building block.

Create the application

1. Open up the command shell or terminal of your choice. You might consider the terminal capabilities in [Visual Studio Code](#). Navigate to the root folder in which you want to build your application. Once there, enter the following command to create a new .NET Console application:

```
dotnet new console -o DaprCounter
```

The command scaffolds a simple “Hello World” .NET Core application.

2. Then, navigate into the new directory created by the previous command:

```
cd DaprCounter
```

3. Run the newly created application using the dotnet run command. Doing so writes “Hello World!” to the console screen:

```
dotnet run
```

Add Dapr State Management

Next, you’ll use the Dapr [state management building block](#) to implement a stateful counter in the program.

You can invoke Dapr APIs across any development platform using Dapr’s native support for HTTP and gRPC. However, .NET Developers will find the Dapr .NET SDK more natural and intuitive. It provides a strongly typed .NET client to call the Dapr APIs. The .NET SDK also tightly integrates with ASP.NET Core.

1. From the terminal window, add the Dapr.Client NuGet package to your application:

```
dotnet add package Dapr.Client
```

Note

If you’re working with a pre-release version of Dapr, be sure to add the `--prerelease` flag to the command.

2. Open the Program.cs file in your favorite editor and update its contents to the following code:

```
using System;
using System.Threading.Tasks;
using Dapr;
using Dapr.Client;

namespace DaprCounter
{
    class Program
    {
        static async Task Main(string[] args)
        {
            var daprClient = new DaprClientBuilder().Build();

            var counter = await daprClient.GetStateAsync<int>("statestore", "counter");

            while (true)
            {
                Console.WriteLine($"Counter = {counter++}");

                await daprClient.SaveStateAsync("statestore", "counter", counter);

                await Task.Delay(1000);
            }
        }
    }
}
```

```
}  
    }  
  }  
}
```

The updated code implements the following steps:

- First a new `DaprClient` instance is instantiated. This class enables you to interact with the Dapr sidecar.
 - From the state store, `DaprClient.GetStateAsync` fetches the value for the counter key. If the key doesn't exist, the default int value (which is 0) is returned.
 - The code then iterates, writing the counter value to the console and saving an incremented value to the state store.
3. The Dapr CLI run command starts the application. It invokes the underlying Dapr runtime and enables both the application and Dapr sidecar to run together. If you omit the app-id, Dapr will generate a unique name for the application. The final segment of the command, `dotnet run`, instructs the Dapr runtime to run the .NET core application.

Important

Care must be taken to always pass an explicit app-id parameter when consuming the state management building block. The block uses the application id value as a *prefix* for its state key for each key/value pair. If the application id changes, you can no longer access the previous stored state.

Now run the application with the following command:

```
dapr run --app-id DaprCounter dotnet run
```

Try stopping and restarting the application. You'll see that the counter doesn't reset. Instead it continues from the previously saved state. The Dapr building block makes the application stateful.

Important

It's important to understand your sample application communicates with a pre-configured state component, but has no direct dependency on it. Dapr abstracts away the dependency. As you'll shortly see, the underlying state store component can be changed with a simple configuration update.

You might be wondering, where exactly is the state stored?

Component configuration files

When you first initialized Dapr for your local environment, it automatically provisioned a Redis container. Dapr then configured the Redis container as the default state store component with a component configuration file, entitled `statestore.yaml`. Here's a look at its contents:

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

```
metadata:
  name: statestore
spec:
  type: state.redis
  version: v1
  metadata:
    - name: redisHost
      value: localhost:6379
    - name: redisPassword
      value: ""
    - name: actorStateStore
      value: "true"
```

Note

Default component configuration files are stored in the \$HOME/.daprc/components folder on Linux/macOS, and in the %USERPROFILE%\daprc\components folder on Windows.

Note the format of the previous component configuration file:

- Each component has a name. In the sample above, the component is named statestore. We used that name in our first code example to tell the Dapr sidecar which component to use.
- Each component configuration file has a spec section. It contains a type field that specifies the component type. The version field specifies the component version. The metadata field contains information that the component requires, such as connection details and other settings. The metadata values will vary for the different types of components.

A Dapr sidecar can consume any Dapr component configured in your application. But, what if you had an architectural justification to limit the accessibility of a component? How could you restrict the Redis component to Dapr sidecars running only in a production environment?

To do so, you could define a namespace for the production environment. You might name it production. In self-hosted mode, you specify the namespace of a Dapr sidecar by setting the NAMESPACE environment variable. When configured, the Dapr sidecar will only load the components that match the namespace. For Kubernetes deployments, the Kubernetes namespace determines the components that are loaded. The following sample shows the Redis component placed in a production namespace. Note the namespace declaration in the metadata element:

```
apiVersion: daprc.io/v1alpha1
kind: Component
metadata:
  name: statestore
  namespace: production
spec:
  type: state.redis
  version: v1
  metadata:
    - name: redisHost
      value: localhost:6379
    - name: redisPassword
      value: ""
    - name: actorStateStore
      value: "true"
```

Important

A namespaced component is only accessible to applications running in the same namespace. If your Dapr application fails to load a component, make sure that the application namespace matches the component namespace. This can be especially tricky in self-hosted mode where the application namespace is stored in a NAMESPACE environment variable.

If needed, you could further restrict a component to a particular application. Within the production namespace, you may want to limit access of the Redis cache to only the DaprCounter application. You do so by specifying scopes in the component configuration. The following example shows how to restrict access to the Redis statestore component to the application DaprCounter in the production namespace:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: statestore
  namespace: production
spec:
  type: state.redis
  version: v1
  metadata:
    - name: redisHost
      value: localhost:6379
    - name: redisPassword
      value: ""
    - name: actorStateStore
      value: "true"
  scopes:
    - DaprCounter
```

Build a multi-container Dapr application

In the first example, you created a simple .NET console application that ran side-by-side with a Dapr sidecar. Modern distributed applications, however, often consist of many moving parts. They can simultaneously run independent microservices. These modern applications are typically containerized and require container orchestration tools such as Docker Compose or Kubernetes.

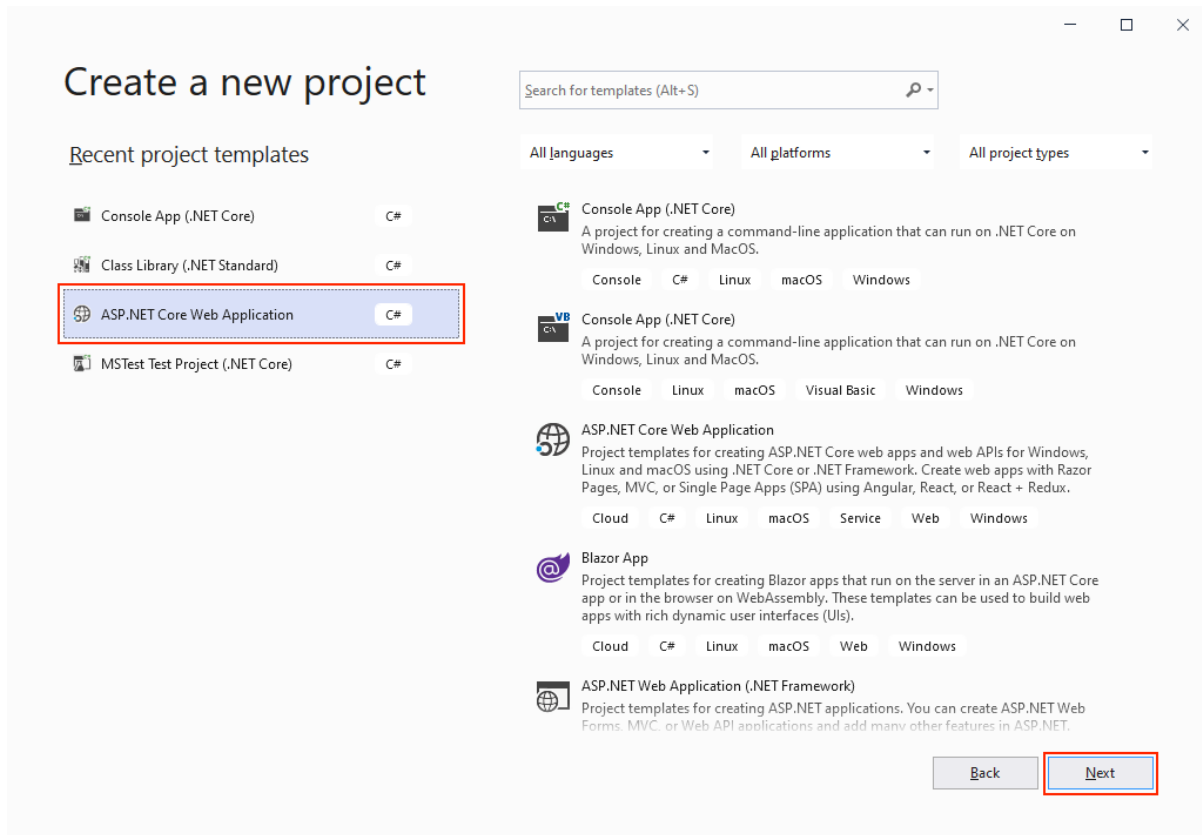
In the next example, you'll create a multi-container application. You'll also use the [Dapr service invocation](#) building block to communicate between services. The solution will consist of a web application that retrieves weather forecasts from a web API. They will each run in a Docker container. You'll use Docker Compose to run the container locally and enable debugging capabilities.

Make sure you've configured your local environment for Dapr and installed the [.NET Core 3 Development Tools](#) (instructions are available at the beginning of this chapter).

Additionally, you'll need complete this sample using [Visual Studio 2019](#) with the **.NET Core cross-platform development** workload installed.

Create the application

1. In Visual Studio 2019, create an **ASP.NET Core Web Application** project:



1. Name your project DaprFrontEnd and your solution DaprMultiContainer:

Configure your new project

ASP.NET Core Web Application Cloud C# Linux macOS Service **Web** Windows

Project name

DaprFrontEnd

Location

C:\Source\Repos\

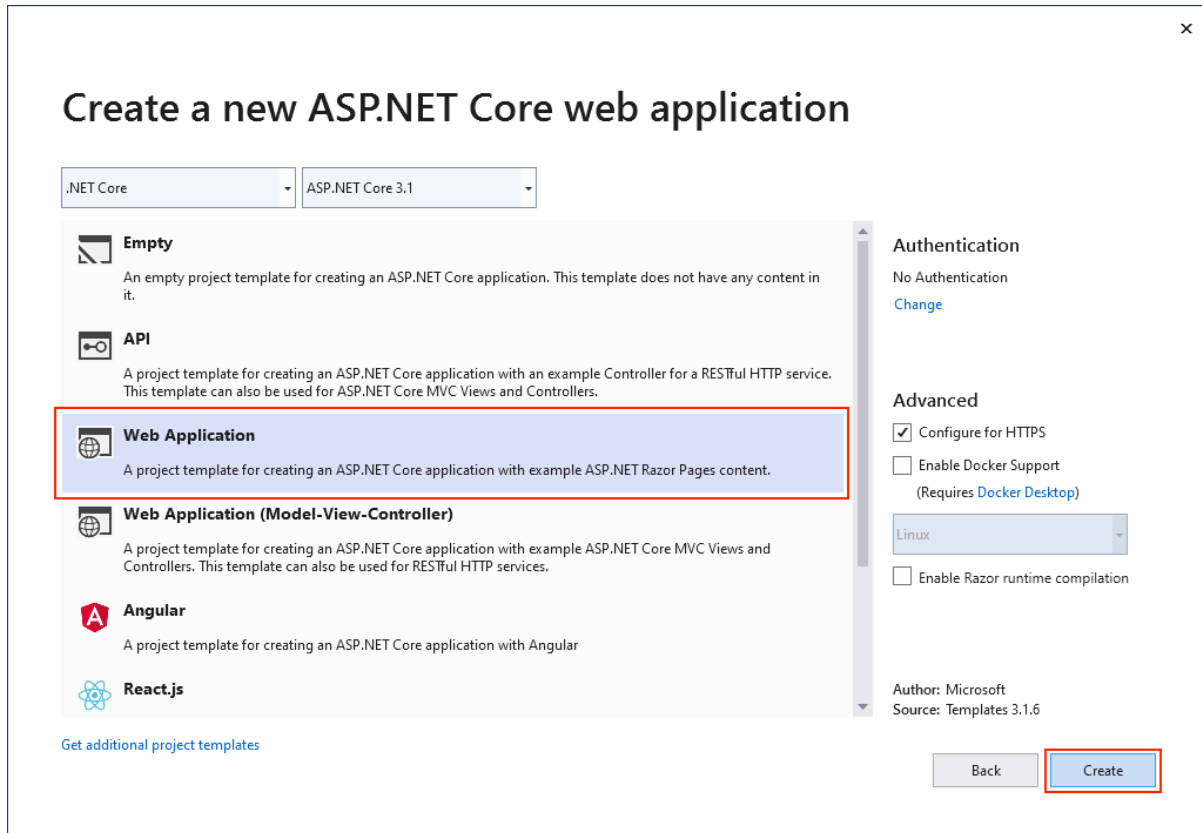
Solution name ⓘ

DaprMultiContainer

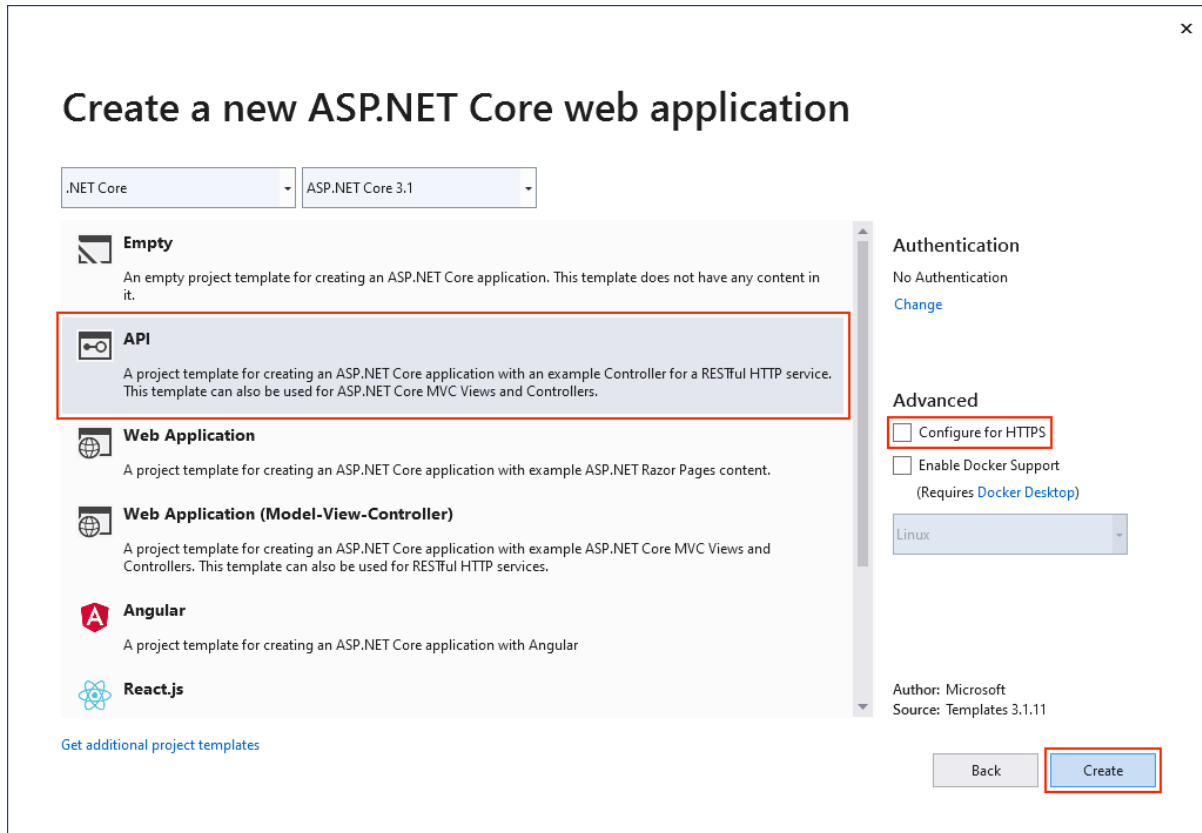
☐ Place solution and project in the same directory

Back **Create**

1. Select **Web Application** to create a web application with Razor pages. Don't select **Enable Docker Support**. You'll add Docker support later.



1. Add a second ASP.NET Core Web Application project to the same solution and call it *DaprBackEnd*. Select **API** as the project type. By default, a Dapr sidecar relies on the network boundary to limit access to its public API. So, clear the checkbox for **Configure for HTTPS**.



Add Dapr service invocation

Now, you'll configure communication between the services using Dapr [service invocation building block](#). You'll enable the web app to retrieve weather forecasts from the web API. The service invocation building block features many benefits. It includes service discovery, automatic retries, message encryption (using mTLS), and improved observability. You'll use the Dapr .NET SDK to invoke the service invocation API on the Dapr sidecar.

1. In Visual Studio, open the Package Manager Console (**Tools > NuGet Package Manager > Package Manager Console**) and make sure that DaprFrontEnd is the default project. From the console, add the Dapr.AspNetCore NuGet package to the project:

Install-Package Dapr.AspNetCore

Note

If you're targeting a version of Dapr.AspNetCore that is in prerelease, you need to specify the -Prerelease flag.

2. In the DaprFrontEnd project, open the *Startup.cs* file, and replace the ConfigureServices method with the following code:

```
// This method gets called by the runtime. Use this method to add services to the
// container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers().AddDapr();
    services.AddRazorPages();
}
```

The call to `AddDapr` registers the `DaprClient` class with the ASP.NET Core dependency injection system. You'll use the `DaprClient` class later on to communicate with the Dapr sidecar.

3. Add a new C# class file named *WeatherForecast* to the *DaprFrontEnd* project:

```
using System;

namespace DaprFrontEnd
{
    public class WeatherForecast
    {
        public DateTime Date { get; set; }

        public int TemperatureC { get; set; }

        public int TemperatureF { get; set; }

        public string Summary { get; set; }
    }
}
```

4. Open the *Index.cshtml.cs* file in the *Pages* folder, and replace its contents with the following code:

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Threading.Tasks;
using Dapr.Client;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace DaprFrontEnd.Pages
{
    public class IndexModel : PageModel
    {
        private readonly DaprClient _daprClient;

        public IndexModel(DaprClient daprClient)
        {
            _daprClient = daprClient ?? throw new
            ArgumentException(nameof(daprClient));
        }

        public async Task OnGet()
        {
            var forecasts = await
            _daprClient.InvokeMethodAsync<IEnumerable<WeatherForecast>>(
                HttpMethod.Get,
                "daprbackend",
                "weatherforecast");
        }
    }
}
```

```

        ViewData["WeatherForecastData"] = forecasts;
    }
}
}

```

You add Dapr capabilities into the web app by injecting the `DaprClient` class into `IndexModel` constructor. In the `OnGet` method, you call the API service with the Dapr service invocation building block. The `OnGet` method is invoked whenever a user visits the home page. You use the `DaprClient.InvokeMethodAsync` method to invoke the `weatherforecast` method of the `daprbackend` service. You'll configure the web API to use `daprbackend` as its application ID later on when configuring it to run with Dapr. Finally, the service response is saved in view data.

5. Replace the contents of the `Index.cshtml` file in the `Pages` folder, with the following code. It displays the weather forecasts stored in the view data to the user:

```

@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

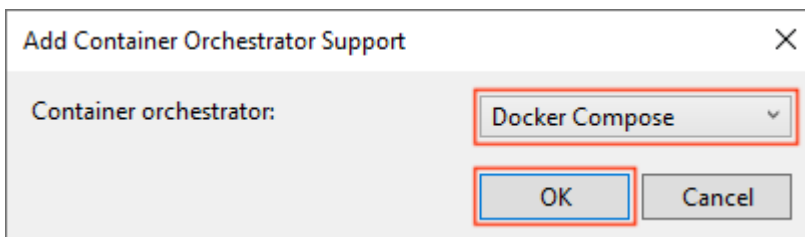
<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with
ASP.NET Core</a>.</p>
    @foreach (var forecast in
(Enumerable<WeatherForecast>)ViewData["WeatherForecastData"])
    {
        <p>The forecast for @forecast.Date is @forecast.Summary!</p>
    }
</div>

```

Add container support

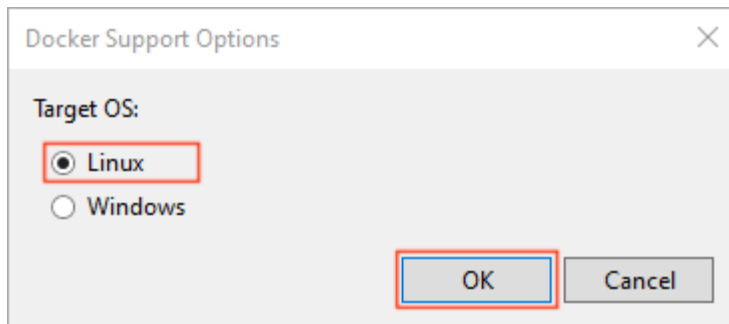
In the final part of this example, you'll add container support and run the solution using Docker Compose.

1. Right-click the `DaprFrontEnd` project, and choose **Add > Container Orchestrator Support**. The **Add Container Orchestrator Support** dialog appears:

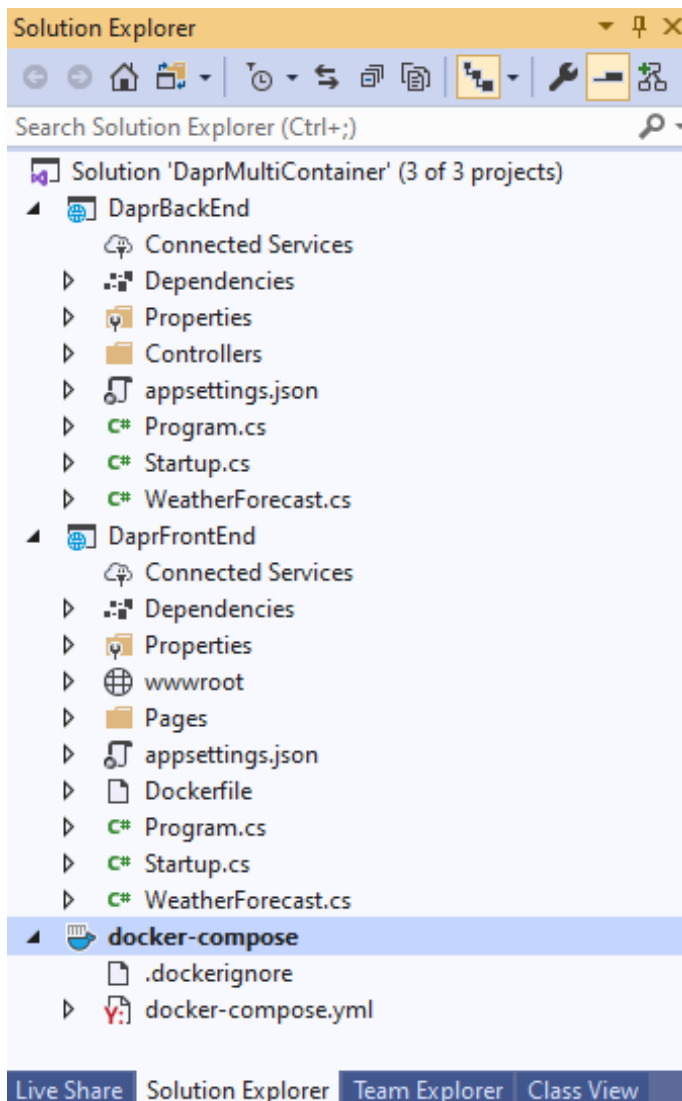


Choose **Docker Compose**.

2. In the next dialog, select **Linux** as the Target OS:



Visual Studio creates a *docker-compose.yml* file and a *.dockerignore* file in the **docker-compose** folder in the solution:



The *docker-compose.yml* file has the following content:

```
version: '3.4'

services:
  daprf frontend:
    image: ${DOCKER_REGISTRY-}daprf frontend
    build:
      context: .
      dockerfile: DaprFrontEnd/Dockerfile
```

The *.dockerignore* file contains file types and extensions that you don't want Docker to include in the container. These files are associated with the development environment and source control and not the app or service you're deploying.

3. In the DaprBackEnd web API project, right-click on the project node, and choose **Add > Container Orchestrator Support**. Choose **Docker Compose**, and then select **Linux** again as the target OS.

Open the *docker-compose.yml* file again and examine its contents. Visual Studio has updated the Docker Compose file. Now both services are included:

```
version: '3.4'

services:
  daprf frontend:
    image: ${DOCKER_REGISTRY-}daprf frontend
    build:
      context: .
      dockerfile: DaprFrontEnd/Dockerfile

  dapr backend:
    image: ${DOCKER_REGISTRY-}dapr backend
    build:
      context: .
      dockerfile: DaprBackEnd/Dockerfile
```

4. To use Dapr building blocks from inside a containerized application, you'll need to add the Dapr sidecars containers to your Compose file. Carefully update the content of the *docker-compose.yml* file to match the following example. Pay close attention to the formatting and spacing and don't use tabs.

```
version: '3.4'

services:
  daprf frontend:
    image: ${DOCKER_REGISTRY-}daprf frontend
    build:
      context: .
      dockerfile: DaprFrontEnd/Dockerfile
    ports:
      - "51000:50001"

  daprf frontend-dapr:
    image: "daprio/daprd:latest"
    command: [ "./daprd", "-app-id", "daprf frontend", "-app-port", "80" ]
    depends_on:
```

```

    - daprf frontend
    network_mode: "service:daprf frontend"

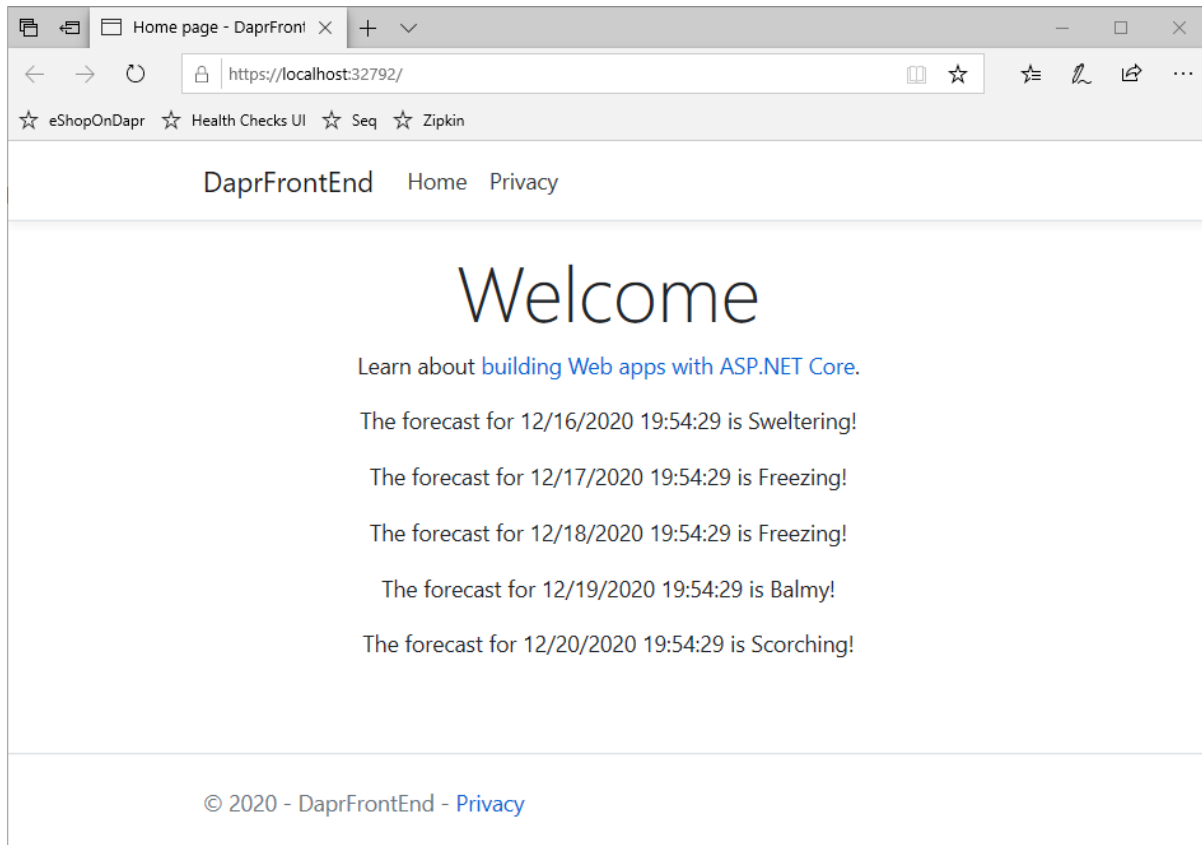
daprbackend:
  image: ${DOCKER_REGISTRY-}daprbackend
  build:
    context: .
    dockerfile: DaprBackend/Dockerfile
  ports:
    - "52000:50001"

daprbackend-dapr:
  image: "daprio/daprd:latest"
  command: [ "./daprd", "-app-id", "daprbackend", "-app-port", "80" ]
  depends_on:
    - daprf frontend
  network_mode: "service:daprbackend"

```

In the updated file, we've added `daprf frontend-dapr` and `daprbackend-dapr` sidecars for the `daprf frontend` and `daprbackend` services respectively. In the updated file, pay close attention to the following changes:

- The sidecars use the `daprio/daprd:latest` container image. The use of the `latest` tag isn't recommended for production scenarios. For production, it's better to use a specific version number.
 - Each service defined in the Compose file has its own network namespace for network isolation purposes. The sidecars use `network_mode: "service:..."` to ensure they run in the same network namespace as the application. Doing so allows the sidecar and the application to communicate using `localhost`.
 - The ports on which the Dapr sidecars are listening for gRPC communication (by default 50001) must be exposed to allow the sidecars to communicate with each other.
5. Run the solution (**F5** or **Ctrl+F5**) to verify that it works as expected. If everything is configured correctly, you should see the weather forecast data:



Running locally with Docker Compose and Visual Studio 2019, you can set breakpoints and debug into the application. For production scenarios, it's recommended to host your application in Kubernetes. This book includes an accompanying reference application, [eShopOnDapr](#), that contains scripts to deploy to Kubernetes.

To learn more about the Dapr service invocation building block used in this walkthrough, refer to [chapter 6](#).

Summary

In this chapter, you had an opportunity to *test drive* Dapr. Using the Dapr .NET SDK, you saw how Dapr integrates with the .NET application platform.

The first example was a simple, stateful, .NET Console application that used the Dapr state management building block.

The second example involved a multi-container application running in Docker. By using Visual Studio with Docker Compose, you experienced the familiar *F5 debugging experience* available across all .NET apps.

You also got a closer look at Dapr component configuration files. They configure the actual infrastructure implementation used by the Dapr building blocks. You can use namespaces and scopes to restrict component access to particular environments and applications.

In the upcoming chapters, you'll dive deep into the building blocks offered by Dapr.

References

- [Dapr documentation - Getting started](#)
- [eShopOnDapr](#)

Dapr reference application

Earlier in the book, you've learned about the foundational benefits of Dapr. You saw how Dapr can help your team construct distributed applications while reducing architectural and operational complexity. Along the way, you've had the opportunity to build some small Dapr apps. Now, it's time to explore an end-to-end microservice application that demonstrates Dapr building blocks and components.

But, first a little history.

eShop on containers

Several years ago, Microsoft, in partnership with leading community experts, released a popular guidance book, entitled [.NET Microservices for Containerized .NET Applications](#). Figure 3-1 shows the book:



Figure 3-1. *.NET Microservices: Architecture for Containerized .NET Applications*.

The book dove deep into the principles, patterns, and best practices for building distributed applications. It included a full-featured microservice reference application that showcased the architectural concepts. Entitled, [eShopOnContainers](#), the application shows an e-Commerce storefront that sells various .NET items, including clothing and coffee mugs. Built in .NET Core, the application is cross-platform and can run in either Linux or Windows containers. Figure 3-2 shows the original eShop architecture.

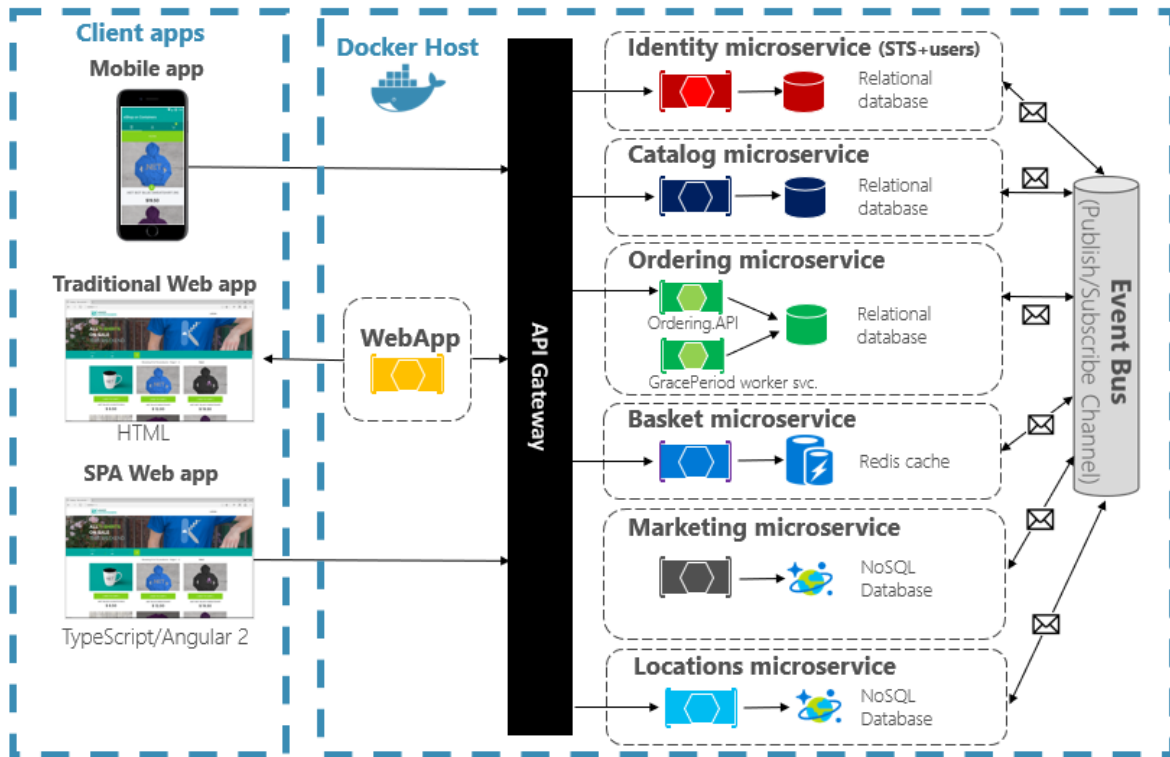


Figure 3-2. Original ShopOnContainers reference application.

As you can see, eShopOnContainers includes many moving parts:

1. Three different front-end clients.
2. An application gateway to abstract the back end from the front end.
3. Several back-end core microservices.
4. An event bus component that enables asynchronous pub/sub messaging.

The eShopOnContainers reference application has been widely accepted across the .NET community and used to model many large commercial microservice applications.

eShop on Dapr

An alternative version of the eShop application accompanies this book. It's called [eShopOnDapr](#). The updated version evolves the earlier eShopOnContainers application by integrating Dapr building blocks and components. Figure 3-3 shows the new streamlined solution architecture:

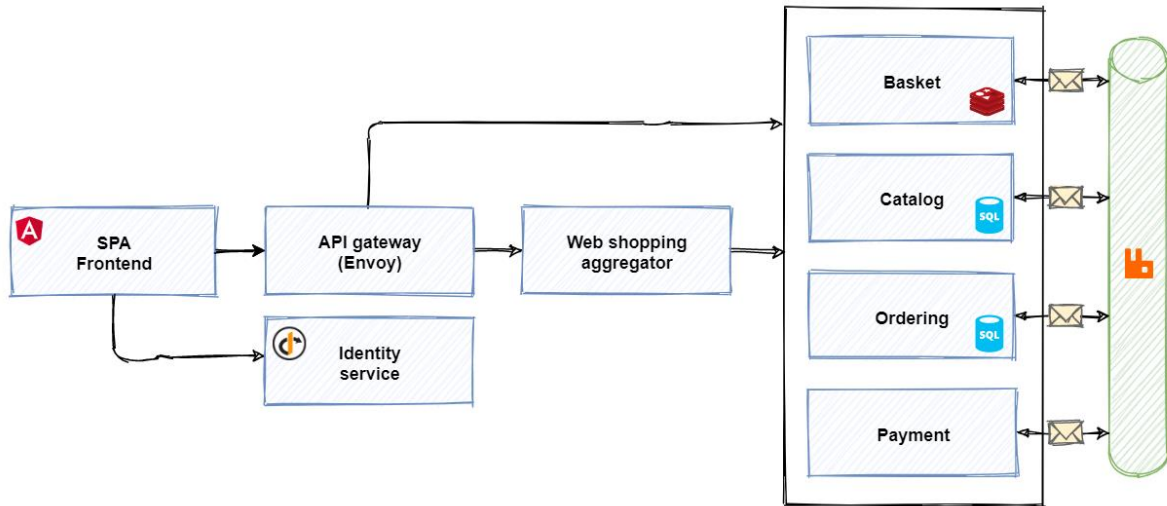


Figure 3-3. eShopOnDapr reference application architecture.

As focus of the eShopOnDapr reference application is on Dapr, the original application has been updated. The architecture consists of:

1. A [Single Page Application](#) front end written in the popular Angular SPA framework. It sends user requests to an API gateway microservice.
2. The API gateway abstracts the back-end core microservices from the front-end client. It's implemented using [Envoy](#), a high performant, open-source service proxy. Envoy routes incoming requests to various back-end microservices. Most requests are simple CRUD operations (for example, get the list of brands from the catalog) and handled by a direct call to a back-end microservice.
3. Other requests are logically more complex and require multiple microservices to work together. For these cases, eShopOnDapr implements an [aggregator microservice](#) that orchestrates a workflow across the microservices needed to complete the operation.
4. The set of core back-end microservices includes functionality required for an e-Commerce store. Each is self-contained and independent of the others. Following widely accepted domain decomposing patterns, each microservice isolates a specific *business capability*:
 - The basket service manages the customer's shopping basket experience.
 - The catalog service manages product items available for sale.
 - The identity service manages authentication and identity.
 - The ordering service handles all aspects of placing and managing orders.
 - The payment service transacts the customer's payment.

Each service has its own persistent storage. Adhering to microservice [best practices](#), there's not a shared datastore with which all services interact.

The design of each microservice is based on its individual requirements. The simple services use basic CRUD operations to access to their underlying data stores. Advanced services, like Ordering, use a Domain-Driven Design approach to manage business complexity. If necessary,

services could be built across different technology stacks, such as .NET Core, Java, Go, NodeJS, and more.

5. Finally, the event bus wraps the Dapr publish/subscribe components. It enables asynchronous publish/subscribe messaging across microservices. Developers can plug in any Dapr-supported message broker.

Application of Dapr building blocks

The eShopOnDapr codebase is more streamlined than the eShopOnContainers codebase. Dapr building blocks replace a large amount of error-prone plumbing code.

Figure 3-4 shows the Dapr integration in the eShop reference application.

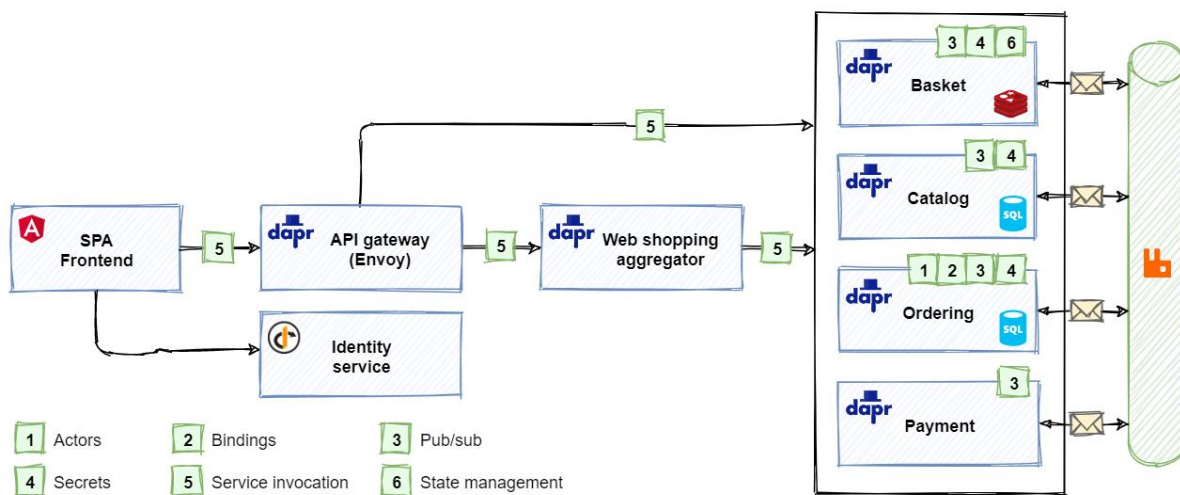


Figure 3-4. Dapr integration in eShopOnDapr.

In the previous figure, you can see which services use which Dapr building blocks.

1. The original eShopOnContainers application demonstrates DDD concepts and patterns in the ordering service. In the updated eShopOnDapr, the ordering service uses the *actor building block* as an alternative implementation. The turn-based access model of actors makes it easy to implement a stateful ordering process with support for cancellation.
2. The ordering service sends order confirmation e-mails using the [bindings building block](#).
3. The back-end services communicate asynchronously using the [publish & subscribe building block](#).
4. Secret management is done by the [secrets building block](#).
5. The API gateway and web shopping aggregator services use the [service invocation building block](#) to invoke methods on the back-end services.
6. The basket service uses the [state management building block](#) to store the state of the customer's shopping basket.

Benefits of applying Dapr to eShop

In general, the use of Dapr building blocks add observability and flexibility to the application:

1. **Observability:** By using the Dapr building blocks, you gain rich distributed tracing for both calls between services and to Dapr components without having to write any code. In eShopOnContainers, a large amount of custom logging is used to provide insight.
2. **Flexibility:** You can now *swap out* infrastructure simply by changing a component configuration file. No code changes are necessary.

Here are some more examples of benefits offered by specific building blocks:

- **Service Invocation**
 - With Dapr’s support for [mTLS](#), services now communicate through encrypted channels.
 - When transient errors occur, service calls are automatically retried.
 - Automatic service discovery reduces the amount of configuration needed for services to find each other.
- **Publish/Subscribe**
 - eShopOnContainer included a large amount of custom code to support both Azure Service Bus and RabbitMQ. Developers used Azure Service Bus for production and RabbitMQ for local development and testing. An IEventBus abstraction layer was created to enable swapping between these message brokers. This layer consisted of approximately *700 lines of error-prone code*. The updated implementation with Dapr requires only *35 lines of code*. That’s **5%** of the original lines of code! More importantly, the implementation is straightforward and easy to understand.
 - eShopOnDapr uses Dapr’s rich ASP.NET Core integration to use pub/sub. You add Topic attributes to ASP.NET Core controller methods to subscribe to messages. Therefore, there’s no need to write a separate message handler loop for each message broker.
 - Messages routed to the service as HTTP calls enable the use of ASP.NET Core middleware to add functionality, without introducing new concepts or SDKs to learn.
- **Bindings**
 - The eShopOnContainers solution contained a *to-do* item for e-mailing an order confirmation to the customer. The thought was to eventually implement a third-party email API such as SendGrid. With Dapr, implementing email notification was as easy as configuring a resource binding. There wasn’t any need to learn external APIs or SDKs.

Note

The Actors building block isn’t covered in the first version of this book. An extensive chapter on the Actor building block and its integration with eShopOnDapr will be included in the 1.1 update.

Summary

In this chapter, you’re introduced to the eShopOnDapr reference application. It’s an evolution of the widely popular eShopOnContainers microservice reference application. eShopOnDapr replaces a large amount of custom functionality with Dapr building blocks and components, dramatically simplifying the complexities required to build a microservices application.

References

- [eShopOnDapr](#)
- [eShopOnContainers](#)
- [.NET Microservices for Containerized .NET Applications](#)
- [Architecting Cloud-Native .NET Apps for Azure](#)

The Dapr state management building block

Distributed applications are composed of independent services. While each service should be stateless, some services must track state to complete business operations. Consider a shopping basket service for an e-Commerce site. If the service can't track state, the customer could lose the shopping basket content by leaving the website, resulting in a lost sale and an unhappy customer experience. For these scenarios, state needs to be persisted to a distributed state store. The [Dapr state management building block](#) simplifies state tracking and offers advanced features across various data stores.

To try out the state management building block, have a look at the [counter application sample in chapter 3](#).

What it solves

Tracking state in a distributed application can be challenging. For example:

- The application may require different types of data stores.
- Different consistency levels may be required for accessing and updating data.
- Multiple users may update data at the same time, requiring conflict resolution.
- Services must retry any short-lived [transient errors](#) that occur while interacting with the data store.

The Dapr state management building block addresses these challenges. It streamlines tracking state without dependencies or a learning curve on third-party storage SDKs.

Important

Dapr state management offers a [key/value](#) API. The feature doesn't support relational or graph data storage.

How it works

The application interacts with a Dapr sidecar to store and retrieve key/value data. Under the hood, the sidecar API consumes a configurable state store component to persist data. Developers can choose from a growing collection of [supported state stores](#) that include Azure Cosmos DB, SQL Server, and Cassandra.

The API can be called with either HTTP or gRPC. Use the following URL to call the HTTP API:

```
http://localhost:<dapr-port>/v1.0/state/<store-name>/
```

- <dapr-port>: the HTTP port that Dapr listens on.
- <store-name>: the name of the state store component to use.

Figure 5-1 shows how a Dapr-enabled shopping basket service stores a key/value pair using the Dapr state store component named `statestore`.

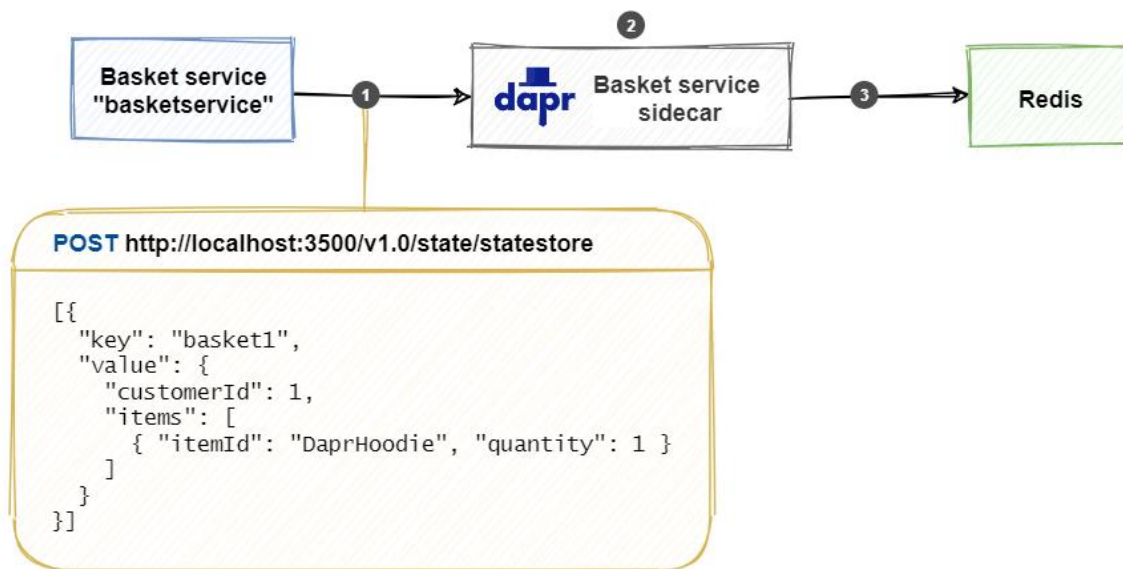


Figure 5-1. Storing a key/value pair in a Dapr state store.

Note the steps in the previous figure:

1. The basket service calls the state management API on the Dapr sidecar. The body of the request encloses a JSON array that can contain multiple key/value pairs.
2. The Dapr sidecar determines the state store based on the component configuration file. In this case, it's a Redis cache state store.
3. The sidecar persists the data to the Redis cache.

Retrieving the stored data is a similar API call. In the example below, a *curl* command retrieves the data by calling the Dapr sidecar API:

```
curl http://localhost:3500/v1.0/state/statestore/basket1
```

The command returns the stored state in the response body:

```
{
  "items": [
    {
      "itemId": "DaprHoodie",
      "quantity": 1
    }
  ],
  "customerId": 1
}
```

The following sections explain how to use the more advanced features of the state management building block.

Consistency

The [CAP theorem](#) is a set of principles that apply to distributed systems that store state. Figure 5-2 shows the three properties of the CAP theorem.

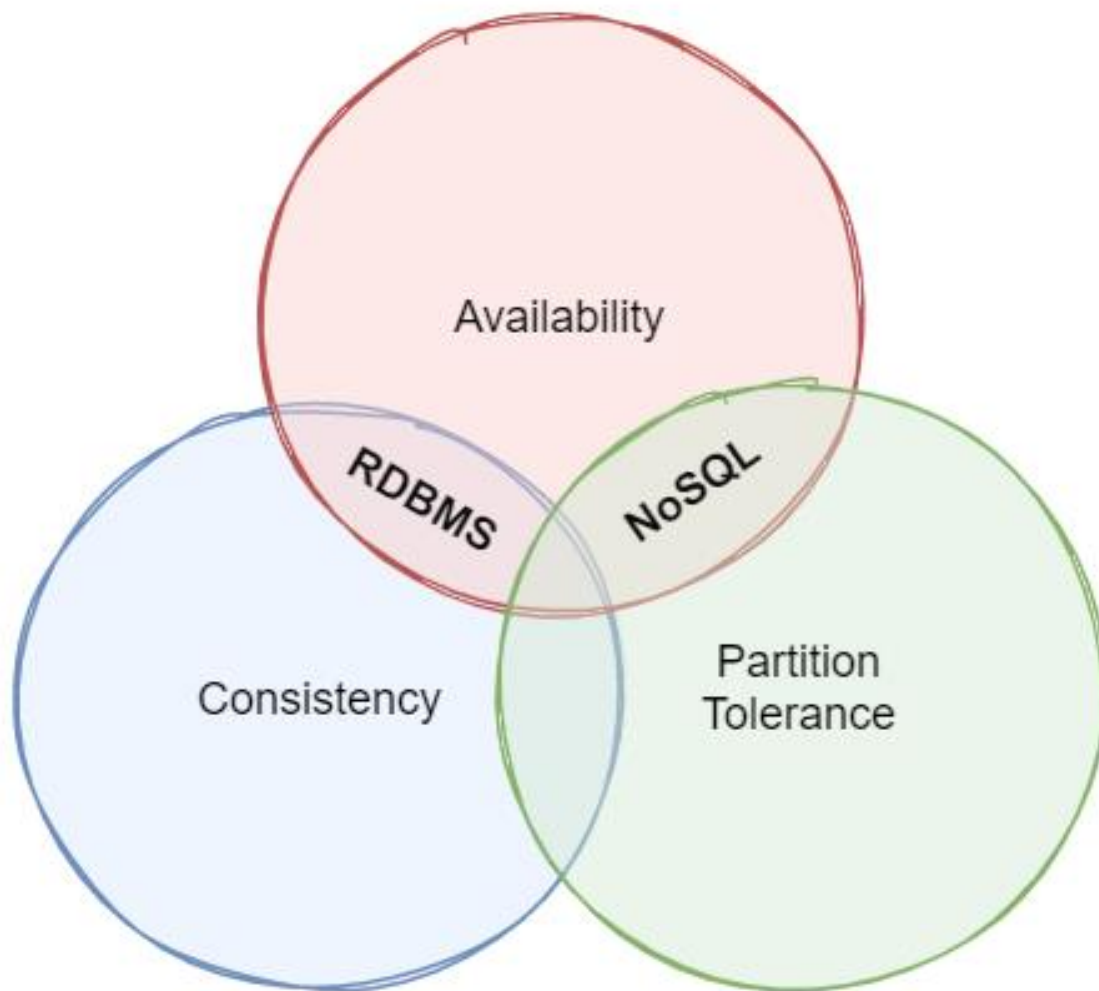


Figure 5-2. The CAP theorem.

The theorem states that distributed data systems offer a trade-off between consistency, availability, and partition tolerance. And, that any datastore can only *guarantee two of the three properties*:

- **Consistency (C)**. Every node in the cluster responds with the most recent data, even if the system must block the request until all replicas update. If you query a “consistent system” for an item that is currently updating, you won’t get a response until all replicas successfully update. However, you’ll always receive the most current data.
- **Availability (A)**. Every node returns an immediate response, even if that response isn’t the most recent data. If you query an “available system” for an item that is updating, you’ll get the best possible answer the service can provide at that moment.
- **Partition Tolerance (P)**. Guarantees the system continues to operate even if a replicated data node fails or loses connectivity with other replicated data nodes.

Distributed applications must handle the **P** property. As services communicate among each other with network calls, network disruptions (**P**) will occur. With that in mind, distributed applications must either be **AP** or **CP**.

AP applications choose availability over consistency. Dapr supports this choice with its **eventual consistency** strategy. Consider an underlying data store, such as Azure CosmosDB, which stores redundant data on multiple replicas. With eventual consistency, the state store writes the update to one replica and completes the write request with the client. After this time, the store will asynchronously update its replicas. Read requests can return data from any of the replicas, including those replicas that haven’t yet received the latest update.

CP applications choose consistency over availability. Dapr supports this choice with its **strong consistency** strategy. In this scenario, the state store will synchronously update *all* (or, in some cases, a *quorum* of) required replicas *before* completing the write request. Read operations will return the most up-to-date data consistently across replicas.

The consistency level for a state operation is specified by attaching a *consistency hint* to the operation. The following *curl* command writes a Hello=World key/value pair to a state store using a strong consistency hint:

```
curl -X POST http://localhost:3500/v1.0/state/<store-name> \
-H "Content-Type: application/json" \
-d '[
  {
    "key": "Hello",
    "value": "World",
    "options": {
      "consistency": "strong"
    }
  }
]'
```

Important

It is up to the Dapr state store component to fulfill the consistency hint attached to the operation. Not all data stores support both consistency levels. If no consistency hint is set, the default behavior is **eventual**.

Concurrency

In a multi-user application, there's a chance that multiple users will update the same data concurrently (at the same time). Dapr supports optimistic concurrency control (OCC) to manage conflicts. OCC is based on an assumption that update conflicts are uncommon because users work on different parts of the data. It's more efficient to assume an update will succeed and retry if it doesn't. The alternative, implementing pessimistic locking, can affect performance with long-running locking causing data contention.

Dapr supports optimistic concurrency control (OCC) using ETags. An ETag is a value associated with a specific version of a stored key/value pair. Each time a key/value pair updates, the ETag value updates as well. When a client retrieves a key/value pair, the response includes the current ETag value. When a client updates or deletes a key/value pair, it must send that ETag value back in the request body. If another client has updated the data in the meantime, the ETags won't match and the request will fail. At this point, the client must retrieve the updated data, make the change again, and resubmit the update. This strategy is called **first-write-wins**.

Dapr also supports a **last-write-wins** strategy. With this approach, the client doesn't attach an ETag to the write request. The state store component will always allow the update, even if the underlying value has changed during the session. Last-write-wins is useful for high-throughput write scenarios with low data contention. As well, overwriting an occasional user update can be tolerated.

Transactions

Dapr can write *multi-item changes* to a data store as a single operation implemented as a transaction. This functionality is only available for data stores that support [ACID](#) transactions. At the time of this writing, these stores include Redis, MongoDB, PostgreSQL, SQL Server, and Azure CosmosDB.

In the example below, a multi-item operation is sent to the state store in a single transaction. All operations must succeed for the transaction to commit. If one or more of the operations fail, the entire transaction rolls back.

```
curl -X POST http://localhost:3500/v1.0/state/<store-name>/transaction \
-H "Content-Type: application/json" \
-d '{
  "operations": [
    {
      "operation": "upsert",
      "request": { "key": "Key1", "value": "Value1" }
    },
    {
      "operation": "delete",
      "request": { "key": "Key2" }
    }
  ]
}
```

```
}']
```

For data stores that don't support transactions, multiple keys can still be sent as a single request. The following example shows a **bulk** write operation:

```
curl -X POST http://localhost:3500/v1.0/state/<store-name> \
-H "Content-Type: application/json" \
-d '[
  { "key": "Key1", "value": "Value1" },
  { "key": "Key2", "value": "Value2" }
]'
```

For bulk operations, Dapr will submit each key/value pair update as a separate request to the data store.

Use the Dapr .NET SDK

The Dapr .NET SDK provides language-specific support for .NET Core platform. Developers can use the `DaprClient` class introduced in [chapter 3](#) to read and write data. The following example shows how to use the `DaprClient.GetStateAsync<TValue>` method to read data from a state store. The method expects the store name, statestore, and key, AMS, as parameters:

```
var weatherForecast = await daprClient.GetStateAsync<WeatherForecast>("statestore", "AMS");
```

If the state store contains no data for key AMS, the result will be default(WeatherForecast).

To write data to the data store, use the `DaprClient.SaveStateAsync<TValue>` method:

```
daprClient.SaveStateAsync("statestore", "AMS", weatherForecast);
```

The example uses the **last-write-wins** strategy as an ETag value isn't passed to the state store component. To use optimistic concurrency control (OCC) with a **first-write-wins** strategy, first retrieve the current ETag using the `DaprClient.GetStateAndETagAsync` method. Then write the updated value and pass along the retrieved ETag using the `DaprClient.TrySaveStateAsync` method.

```
var (weatherForecast, etag) = await
daprClient.GetStateAndETagAsync<WeatherForecast>("statestore", city);

// ... make some changes to the retrieved weather forecast

var result = await daprClient.TrySaveStateAsync("statestore", city, weatherForecast, etag);
```

The `DaprClient.TrySaveStateAsync` method fails when the data (and associated ETag) has been changed in the state store after the data was retrieved. The method returns a boolean value to indicate whether the call succeeded. A strategy to handle the failure is to simply reload the updated data from the state store, make the change again, and resubmit the update.

If you always want a write to succeed regardless of other changes to the data, use the **last-write-wins** strategy.

The SDK provides other methods to retrieve data in bulk, delete data, and execute transactions. For more information, see the [Dapr .NET SDK repository](#).

ASP.NET Core integration

Dapr also supports ASP.NET Core, a cross-platform framework for building modern cloud-based web applications. The Dapr SDK integrates state management capabilities directly into the [ASP.NET Core model binding](#) capabilities. Configuration is simple. Add the `IMVCBuilder.AddDapr` by appending the `.AddDapr` extension method in your `Startup.cs` class as shown in the next example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers().AddDapr();
}
```

Once configured, Dapr can inject a key/value pair directly into a controller action using the ASP.NET Core `FromState` attribute. Referencing the `DaprClient` object is no longer necessary. The next example shows a Web API that returns the weather forecast for a given city:

```
[HttpGet("{city}")]
public ActionResult<WeatherForecast> Get([FromState("statestore", "city")]
StateEntry<WeatherForecast> forecast)
{
    if (forecast.Value == null)
    {
        return NotFound();
    }

    return forecast.Value;
}
```

In the example, the controller loads the weather forecast using the `FromState` attribute. The first attribute parameter is the state store, `statestore`. The second attribute parameter, `city`, is the name of the [route template](#) variable to get the state key. If you omit the second parameter, the name of the bound method parameter (`forecast`) is used to look up the route template variable.

The `StateEntry` class contains properties for all the information that is retrieved for a single key/value pair: `StoreName`, `Key`, `Value`, and `ETag`. The `ETag` is useful for implementing optimistic concurrency control (OCC) strategy. The class also provides methods to delete or update retrieved key/value data without requiring a `DaprClient` instance. In the next example, the `TrySaveAsync` method is used to update the retrieved weather forecast using OCC.

```
[HttpPut("{city}")]
public async Task Put(WeatherForecast updatedForecast, [FromState("statestore", "city")]
StateEntry<WeatherForecast> currentForecast)
{
    // update cached current forecast with updated forecast passed into service endpoint
    currentForecast.Value = updatedForecast;

    // update state store
    var success = await currentForecast.TrySaveAsync();

    // ... check result
}
```

State store components

At the time of this writing, Dapr provides support for the following transactional state stores:

- Azure CosmosDB
- Azure SQL Server
- MongoDB
- PostgreSQL
- Redis

Dapr also includes support for state stores that support CRUD operations, but not transactional capabilities:

- Aerospike
- Azure Blob Storage
- Azure Table Storage
- Cassandra
- Cloudstate
- Couchbase
- etcd
- Google Cloud Firestore
- Hashicorp Consul
- Hazelcast
- Memcached
- Zookeeper

Configuration

When initialized for local, self-hosted development, Dapr registers Redis as the default state store. Here's an example of the default state store configuration. Note the default name, `statestore`:

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

Note

Many state stores can be registered to a single application each with a different name.

The Redis state store requires `redisHost` and `redisPassword` metadata to connect to the Redis instance. In the example above, the Redis password (which is an empty string by default) is stored as a plain string. The best practice is to avoid clear-text strings and always use secret references. To learn more about secret management, see [chapter 10](#).

The other metadata field, `actorStateStore`, indicates whether the state store can be consumed by the actors building block.

Key prefix strategies

State store components enable different strategies to store key/value pairs in the underlying store. Recall the earlier example of a shopping basket service storing items a customer wishes to purchase:

```
curl -X POST http://localhost:3500/v1.0/state/statestore \
-H "Content-Type: application/json" \
-d '{
  "key": "basket1",
  "value": {
    "customerId": 1,
    "items": [
      { "itemId": "DaprHoodie", "quantity": 1 }
    ]
  }
}'
```

Using the Redis Console tool, look inside the Redis cache to see how the Redis state store component persisted the data:

```
127.0.0.1:6379> KEYS *
1) "basketservice||basket1"

127.0.0.1:6379> HGETALL basketservice||basket1
1) "data"
2) "{\"items\": [{\"itemId\": \"DaprHoodie\", \"quantity\": 1}], \"customerId\": 1}"
3) "version"
4) "1"
```

The output shows the full Redis **key** for the data as `basketservice||basket1`. By default, Dapr uses the application id of the Dapr instance (`basketservice`) as a prefix for the key. This naming convention enables multiple Dapr instances to share the same data store without key name collisions. For the developer, it's critical always to specify the same application id when running the application with Dapr. If omitted, Dapr will generate a unique application ID. If the application id changes, the application can no longer access the state stored with the previous key prefix.

That said, it's possible to configure a *constant value* for the key prefix in the `keyPrefix` metadata field in the state store component file. Consider the following example:

```
spec:
  metadata:
    - name: keyPrefix
    - value: MyPrefix
```


A constant key prefix enables the state store to be accessed across multiple Dapr applications. What's more, setting the keyPrefix to none omits the prefix completely.

Reference application: eShopOnDapr

This book includes a reference application entitled eShopOnDapr. It's modeled from an earlier Microsoft microservices reference application, eShopOnContainers.

The original [eShopOnContainers](#) architecture used an IBasketRepository interface to read and write data for the basket service. The RedisBasketRepository class provided the implementation using Redis as the underlying data store:

```
public class RedisBasketRepository : IBasketRepository
{
    private readonly ConnectionMultiplexer _redis;
    private readonly IDatabase _database;

    public RedisBasketRepository(ConnectionMultiplexer redis)
    {
        _redis = redis;
        _database = redis.GetDatabase();
    }

    public async Task<CustomerBasket> GetBasketAsync(string customerId)
    {
        var data = await _database.StringGetAsync(customerId);

        if (data.IsNullOrEmpty())
        {
            return null;
        }

        return JsonConvert.DeserializeObject<CustomerBasket>(data);
    }

    // ...
}
```

This code uses the third-party StackExchange.Redis NuGet package. The following steps are required to load the shopping basket for a given customer:

1. Inject a ConnectionMultiplexer into the constructor. The ConnectionMultiplexer is registered with the dependency injection framework in the Startup.cs file:

```
services.AddSingleton<ConnectionMultiplexer>(sp =>
{
    var settings = sp.GetRequiredService<IOptions<BasketSettings>>().Value;
    var configuration = ConfigurationOptions.Parse(settings.ConnectionString, true);
    configuration.ResolveDns = true;
    return ConnectionMultiplexer.Connect(configuration);
});
```

2. Use the ConnectionMultiplexer to create an IDatabase instance in each consuming class.

3. Use the IDatabase instance to execute a Redis StringGet call using the given customerId as the key.
4. Check if data is loaded from Redis; if not, return null.
5. Deserialize the data from Redis to a CustomerBasket object and return the result.

In the updated [eShopOnDapr](#) reference application, a new DaprBasketRepository class replaces the RedisBasketRepository class:

```
public class DaprBasketRepository : IBasketRepository
{
    private const string StoreName = "eshop-basket-statestore";

    private readonly DaprClient _daprClient;

    public DaprBasketRepository(DaprClient daprClient)
    {
        _daprClient = daprClient ?? throw new ArgumentNullException(nameof(daprClient));
    }

    public async Task<CustomerBasket> GetBasketAsync(string customerId)
    {
        return await _daprClient.GetStateAsync<CustomerBasket>(StoreName, customerId);
    }

    // ...
}
```

The updated code uses the Dapr .NET SDK to read and write data using the state management building block. The new steps to load the basket for a customer are dramatically simplified:

1. Inject a DaprClient into the constructor. The DaprClient is registered with the dependency injection framework in the Startup.cs file.
2. Use the DaprClient.GetStateAsync method to load the customer's shopping basket items from the configured state store and return the result.

The updated implementation still uses Redis as the underlying data store. But, Dapr abstracts the StackExchange.Redis references and complexity from the application. A Dapr configuration file is all that's needed:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: eshop-basket-statestore
  namespace: eshop
spec:
  type: state.redis
  version: v1
  metadata:
    - name: redisHost
      value: redis:6379
    - name: redisPassword
      secretKeyRef:
        name: redisPassword
  auth:
    secretStore: eshop-secretstore
```

The Dapr implementation also simplifies changing the underlying data store. For example, switching to Azure Table Storage requires only changing the contents of the configuration file. No code changes are necessary.

Summary

The Dapr state management building block offers an API for storing key/value data across various data stores. The API provides support for:

- Bulk operations
- Strong and eventual consistency
- Optimistic concurrency control
- Multi-item transactions

The .NET SDK provides language-specific support for .NET Core and ASP.NET Core. Model binding integration simplifies accessing and updating state from ASP.NET Core controller action methods.

In the eShopOnDapr reference application, the benefits to moving to Dapr state management are clear:

1. The new implementation uses fewer lines of code.
2. It abstracts away the complexity of the third-party StackExchange.Redis API.
3. Replacing the underlying Redis cache with a different type of data store now only requires changes to the state store configuration file.

References

- [Dapr supported state stores](#)

The Dapr service invocation building block

Across a distributed system, one service often needs to communicate with another to complete a business operation. The [Dapr service invocation building block](#) can help streamline the communication between services.

What it solves

Making calls between services in a distributed application may appear easy, but there are many challenges involved. For example:

- Where the other services are located.
- How to call a service securely, given the service address.
- How to handle retries when short-lived [transient errors](#) occur.

Lastly, as distributed applications compose many different services, capturing insights across service call graphs are critical to diagnosing production issues.

The service invocation building block addresses these challenges by using a Dapr sidecar as a [reverse proxy](#) for your service.

How it works

Let's start with an example. Consider two services, "Service A" and "Service B". Service A needs to call the catalog/items API on Service B. While Service A could take a dependency on Service B and make a direct call to it, Service A instead invokes the service invocation API on the Dapr sidecar. Figure 6-1 shows the operation.

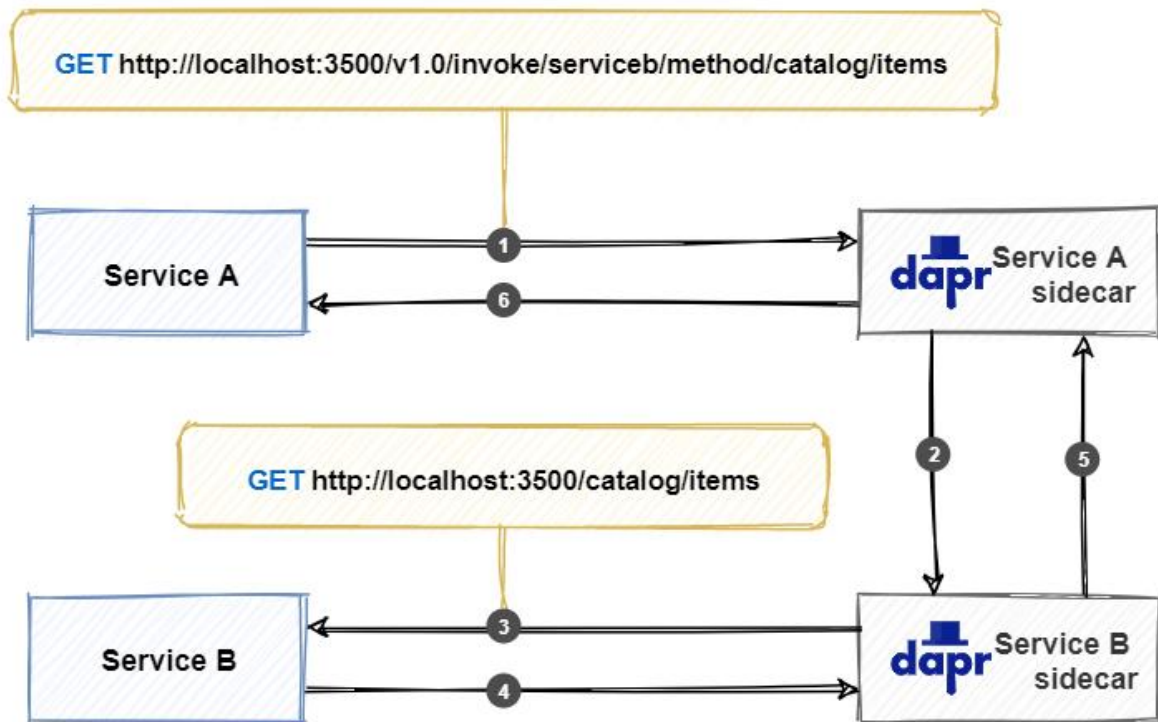


Figure 6-1. How Dapr service invocation works.

Note the steps from the previous figure:

1. Service A makes a call to the catalog/items endpoint in Service B by invoking the service invocation API on the Service A sidecar.

Note

The sidecar uses a pluggable name resolution mechanism to resolve the address of Service B. In self-hosted mode, Dapr uses [mDNS](#) to find it. When running in Kubernetes mode, the Kubernetes DNS service determines the address.

2. The Service A sidecar forwards the request to the Service B sidecar.
3. The Service B sidecar makes the actual catalog/items request against the Service B API.
4. Service B executes the request and returns a response back to its sidecar.
5. The Service B sidecar forwards the response back to the Service A sidecar.
6. The Service A sidecar returns the response back to Service A.

Because the calls flow through sidecars, Dapr can inject some useful cross-cutting behaviors:

- Automatically retry calls upon failure.
- Make calls between services secure with mutual (mTLS) authentication, including automatic certificate rollover.

- Control what operations clients can do using access control policies.
- Capture traces and metrics for all calls between services to provide insights and diagnostics.

Any application can invoke a Dapr sidecar by using the native **invoke** API built into Dapr. The API can be called with either HTTP or gRPC. Use the following URL to call the HTTP API:

```
http://localhost:<dapr-port>/v1.0/invoke/<application-id>/method/<method-name>
```

- <dapr-port> the HTTP port that Dapr is listening on.
- <application-id> application ID of the service to call.
- <method-name> name of the method to invoke on the remote service.

In the following example, a *curl* call is made to the catalog/items 'GET' endpoint of Service B:

```
curl http://localhost:3500/v1.0/invoke/serviceb/method/catalog/items
```

Note

The Dapr APIs enable any application stack that supports HTTP or gRPC to use Dapr building blocks. Therefore, the service invocation building block can act as a bridge between protocols. Services can communicate with each other using HTTP, gRPC or a combination of both.

In the next section, you'll learn how to use the .NET SDK to simplify service invocation calls.

Use the Dapr .NET SDK

The Dapr [.NET SDK](#) provides .NET developers with an intuitive and language-specific way to interact with Dapr. The SDK offers developers three ways of making remote service invocation calls:

1. Invoke HTTP services using `HttpClient`
2. Invoke HTTP services using `DaprClient`
3. Invoke gRPC services using `DaprClient`

Invoke HTTP services using `HttpClient`

The preferred way to call an HTTP endpoint is to use Dapr's rich integration with `HttpClient`. The following example submits an order by calling the `submit` method of the `orderservice` application:

```
var httpClient = DaprClient.CreateHttpClient();
await httpClient.PostAsJsonAsync("http://orderservice/submit", order);
```

In the example, `DaprClient.CreateHttpClient` returns an `HttpClient` instance that is used to perform Dapr service invocation. The returned `HttpClient` uses a special Dapr message handler that rewrites URLs of outgoing requests. The host name is interpreted as the application ID of the service to call. The rewritten request that's actually being called is:

```
http://127.0.0.1:3500/v1/invoke/orderservice/method/submit
```

This example uses the default value for the Dapr HTTP endpoint, which is `http://127.0.0.1:<dapr-http-port>/`. The value of `dapr-http-port` is taken from the `DAPR_HTTP_PORT` environment variable. If it's not set, the default port number 3500 is used.

Alternatively, you can configure a custom endpoint in the call to `DaprClient.CreateHttpClient`:

```
var httpClient = DaprClient.CreateHttpClient(daprEndpoint = "localhost:4000");
```

You can also directly set the base address by specifying the application ID. This makes it possible to use relative URIs when making a call:

```
var httpClient = DaprClient.CreateHttpClient("orderservice");
await httpClient.PostAsJsonAsync("/submit");
```

The `HttpClient` object is intended to be long-lived. A single `HttpClient` instance can be reused for the lifetime of the application. The next scenario demonstrates how an `OrderServiceClient` class reuses a Dapr `HttpClient` instance:

```
public void ConfigureServices(IServiceCollection services)
{
    // ...
    services.AddSingleton<IOrderServiceClient, OrderServiceClient>(
        _ => new OrderServiceClient(DaprClient.CreateInvokeHttpClient("orderservice")));
}
```

In the snippet above, the `OrderServiceClient` is registered as a singleton with the ASP.NET Core dependency injection system. An implementation factory creates a new `HttpClient` instance by calling `DaprClient.CreateInvokeHttpClient`. It then uses the newly created `HttpClient` to instantiate the `OrderServiceClient` object. By registering the `OrderServiceClient` as a singleton, it will be reused for the lifetime of the application.

The `OrderServiceClient` itself has no Dapr-specific code. Even though Dapr service invocation is used under the hood, you can treat the Dapr `HttpClient` like any other `HttpClient`:

```
public class OrderServiceClient : IOrderServiceClient
{
    private readonly HttpClient _httpClient;

    public OrderServiceClient(HttpClient httpClient)
    {
        _httpClient = httpClient ?? throw new ArgumentNullException(nameof(httpClient));
    }

    public async Task SubmitOrder(Order order)
    {
        var response = await _httpClient.PostAsJsonAsync("submit", order);
        response.EnsureSuccessStatusCode();
    }
}
```

Using the `HttpClient` class with Dapr service invocation has many benefits:

- `HttpClient` is a well-known class that many developers already use in their code. Using `HttpClient` for Dapr service invocation allows developers to reuse their existing skills.

- HttpClient supports advanced scenarios, such as custom headers, and full control over request and response messages.
- In .NET 5, HttpClient supports automatic serialization and deserialization using System.Text.Json.
- HttpClient integrates with many existing frameworks and libraries, such as [Refit](#), [RestSharp](#), and [Polly](#).

Invoke HTTP services using DaprClient

While HttpClient is the preferred way to invoke services using HTTP semantics, you can also use the DaprClient.InvokeMethodAsync family of methods. The following example submits an order by calling the submit method of the orderservice application:

```
var daprClient = new DaprClientBuilder().Build();
try
{
    var confirmation =
        await daprClient.InvokeMethodAsync<Order, OrderConfirmation>(
            "orderservice", "submit", order);
}
catch (InvocationException ex)
{
    // Handle error
}
```

The third argument, an order object, is serialized internally (with System.Text.JsonSerializer) and sent as the request payload. The .NET SDK takes care of the call to the sidecar. It also deserializes the response to an OrderConfirmation object. Because no HTTP method is specified, the request is executed as an HTTP POST.

The next example demonstrates how you can make an HTTP GET request by specifying the HttpMethod:

```
var catalogItems = await
    daprClient.InvokeMethodAsync<IEnumerable<CatalogItem>>(HttpMethod.Get, "catalogservice",
        "items");
```

For some scenarios, you may require more control over the request message. For example, when you need to specify request headers, or you want to use a custom serializer for the payload. DaprClient.CreateInvokeMethodRequest creates an HttpRequestMessage. The following example demonstrates how to add an HTTP authorization header to a request message:

```
var request = daprClient.CreateInvokeMethodRequest("orderservice", "submit", order);
request.Headers.Authorization = new AuthenticationHeaderValue("bearer", token);
```

The HttpRequestMessage now has the following properties set:

- Url = http://127.0.0.1:3500/v1.0/invoke/orderservice/method/submit
- HttpMethod = POST
- Content = JsonContent object containing the JSON-serialized order
- Headers.Authorization = "bearer <token>"

Once you've got the request set up the way you want, use DaprClient.InvokeMethodAsync to send it:


```
var orderConfirmation = await daprClient.InvokeMethodAsync<OrderConfirmation>(request);
```

DaprClient.InvokeMethodAsync deserializes the response to an OrderConfirmation object if the request is successful. Alternatively, you can use DaprClient.InvokeMethodWithResponseAsync to get full access to the underlying HttpResponseMessage:

```
var response = await daprClient.InvokeMethodWithResponseAsync(request);
response.EnsureSuccessStatusCode();

var orderConfirmation = response.Content.ReadFromJsonAsync<OrderConfirmation>();
```

Note

For service invocation calls using HTTP, it's worth considering using the Dapr HttpClient integration presented in the previous section. Using HttpClient gives you additional benefits such as integration with existing frameworks and libraries.

Invoke gRPC services using DaprClient

DaprClient provides a family of InvokeMethodGrpcAsync methods for calling gRPC endpoints. The main difference with the HTTP methods is the use of a Protobuf serializer instead of JSON. The following example invokes the submitOrder method of the orderservice over gRPC.

```
var daprClient = new DaprClientBuilder().Build();
try
{
    var confirmation = await daprClient.InvokeMethodGrpcAsync<Order,
    OrderConfirmation>("orderservice", "submitOrder", order);
}
catch (InvocationException ex)
{
    // Handle error
}
```

In the example above, DaprClient serializes the given order object using [Protobuf](#) and uses the result as the gRPC request body. Likewise, the response body is Protobuf deserialized and returned to the caller. Protobuf typically provides better performance than the JSON payloads used in HTTP service invocation.

Reference application: eShopOnDapr

The original [eShopOnContainers](#) microservice reference architecture from Microsoft used a mix of HTTP/REST and gRPC services. The use of gRPC was limited to communication between an [aggregator service](#) and core back-end services. Figure 6-2 show the architecture:

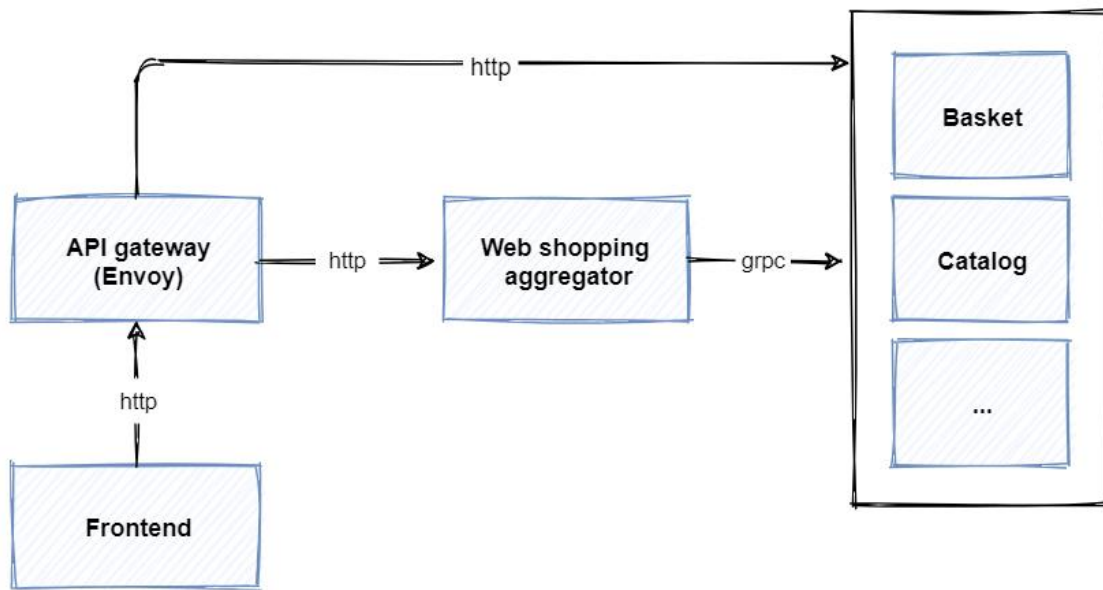


Figure 6-2. gRPC and HTTP/REST calls in eShopOnContainers.

Note the steps from the previous figure:

1. The front end calls the [API gateway](#) using HTTP/REST.
2. The API gateway forwards simple [CRUD](#) (Create, Read, Update, Delete) requests directly to a core back-end service using HTTP/REST.
3. The API gateway forwards complex requests that involve coordinated calls to multiple back-end services to the web shopping aggregator service.
4. The aggregator service uses gRPC to call core back-end services.

In the recently updated eShopOnDapr implementation, Dapr sidecars are added to the services and API gateway. Figure 6-3 show the updated architecture:

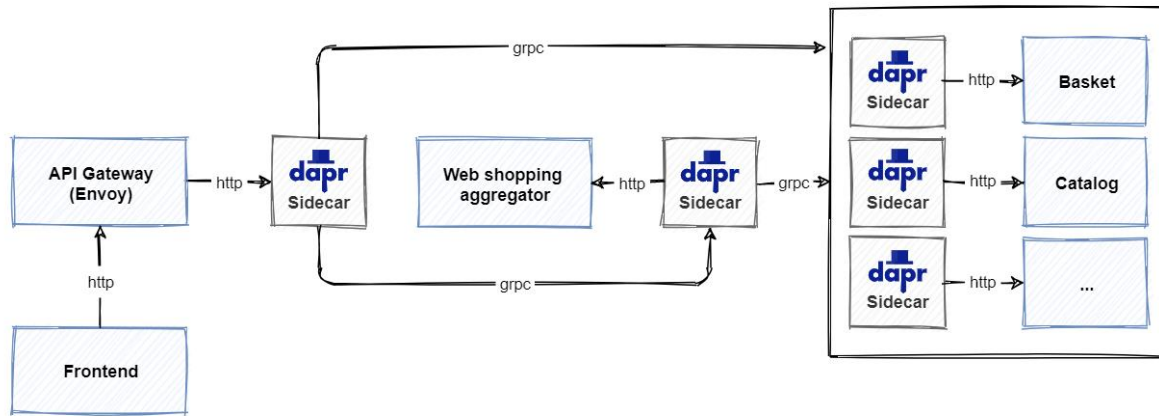


Figure 6-3. Updated eShop architecture using Dapr.

Note the updated steps from the previous figure:

1. The front end still uses HTTP/REST to call the API gateway.
2. The API gateway forwards HTTP requests to its Dapr sidecar.
3. The API gateway sidecar sends the request to the sidecar of the aggregator or back-end service.
4. The aggregator service uses the Dapr .NET SDK to call back-end services through their sidecar architecture.

Dapr implements calls between sidecars with gRPC. So even if you're invoking a remote service with HTTP/REST semantics, a part of the transport is still implemented using gRPC.

The eShopOnDapr reference application benefits from the Dapr service invocation building block. The benefits include service discovery, automatic mTLS, and observability.

Forward HTTP requests using Envoy and Dapr

Both the original and updated eShop application leverage the [Envoy proxy](#) as an API gateway. Envoy is an open-source proxy and communication bus that is popular across modern distributed applications. Originating from Lyft, Envoy is owned and maintained by the [Cloud-Native Computing Foundation](#).

In the original eShopOnContainers implementation, the Envoy API gateway forwarded incoming HTTP requests directly to aggregator or back-end services. In the new eShopOnDapr, the Envoy proxy forwards the request to a Dapr sidecar. The sidecar provides service invocation, mTLS, and observability.

Envoy is configured using a YAML definition file to control the proxy's behavior. To enable Envoy to forward HTTP requests to a Dapr sidecar container, a dapr cluster is added to the configuration. The cluster configuration contains a host that points to the HTTP port on which the Dapr sidecar is listening:

```
clusters:
- name: dapr
  connect_timeout: 0.25s
  type: strict_dns
  hosts:
  - socket_address:
      address: 127.0.0.1
      port_value: 3500
```

The Envoy routes configuration is updated to rewrite incoming requests as calls to the Dapr sidecar (pay close attention to the `prefix_rewrite` key/value pair):

```
- name: "c-short"
  match:
    prefix: "/c/"
  route:
    auto_host_rewrite: true
    prefix_rewrite: "/v1.0/invoke/catalog-api/method/"
    cluster: dapr
```

Consider a scenario where the front-end client wants to retrieve a list of catalog items. The Catalog API provides an endpoint for getting the catalog items:

```
[Route("api/v1/[controller]")]
[ApiController]
public class CatalogController : ControllerBase
{
    [HttpGet("items")]
    public async Task<IActionResult> ItemsAsync(
        [FromQuery] int pageSize = 10,
        [FromQuery] int pageIndex = 0)
    {
        // ...
    }
}
```

First, the front end makes a direct HTTP call to the Envoy API gateway.

```
GET http://<api-gateway>/c/api/v1/catalog/items?pageSize=20
```

The Envoy proxy matches the route, rewrites the HTTP request, and forwards it to the `invoke` API of its Dapr sidecar:

```
GET http://127.0.0.1:3500/v1.0/invoke/catalog-api/method/api/v1/catalog/items?pageSize=20
```

The sidecar handles service discovery and routes the request to the Catalog API sidecar. Finally, the sidecar calls the Catalog API to execute the request, fetch catalog items, and return a response:

```
GET http://localhost/api/v1/catalog/items?pageSize=20
```

Make aggregated service calls using the .NET SDK

Most calls from the eShop front end are simple CRUD calls. The API gateway forwards them to a single service for processing. Some scenarios, however, require multiple back-end services to work together to complete a request. For these more complex calls, eShop uses the web shopping aggregator service to mediate the workflow across multiple services. Figure 6-4 show the processing sequence of adding an item to your shopping basket:

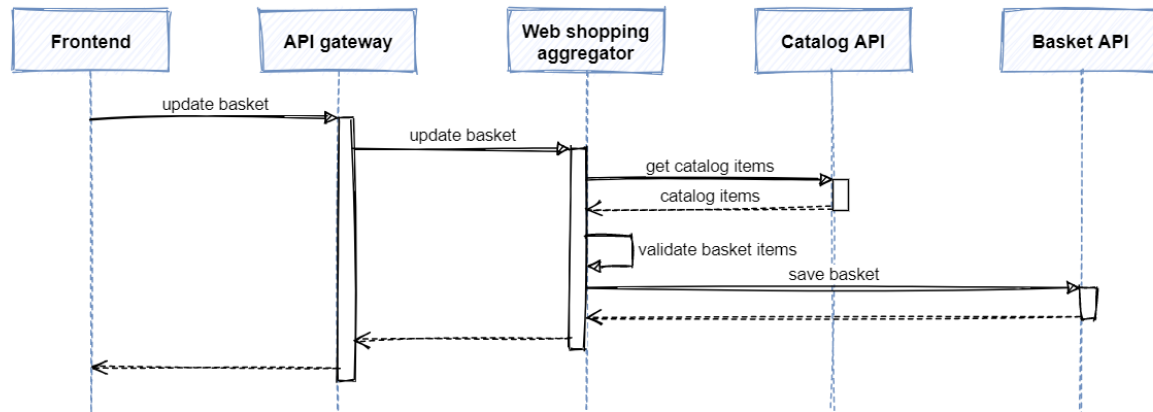


Figure 6-4. Update shopping basket sequence.

The aggregator service first retrieves catalog items from the Catalog API. It then validates item availability and pricing. Finally, the aggregator service saves the updated shopping basket by calling the Basket API.

The aggregator service contains a `BasketController` that provides an endpoint for updating the shopping basket:

```

[Route("api/v1/[controller]")]
[Authorize]
[ApiController]
public class BasketController : ControllerBase
{
    private readonly ICatalogService _catalog;
    private readonly IBasketService _basket;

    [HttpPost]
    [HttpPut]
    public async Task<ActionResult<BasketData>> UpdateAllBasketAsync(
        [FromBody] UpdateBasketRequest data, [FromHeader] string authorization)
    {
        // Get the item details from the catalog API.
        var catalogItems = await _catalog.GetCatalogItemsAsync(
            data.Items.Select(x => x.ProductId));

        // Check item availability and prices; store results in basket object.
        var basket = CreateValidatedBasket(data, catalogItems);

        // Save the shopping basket.
        await _basket.UpdateAsync(basket, authorization);

        return basket;
    }

    // ...
}
  
```

The `UpdateAllBasketAsync` method gets the `Authorization` header of the incoming request using a `FromHeader` attribute. The `Authorization` header contains the access token that is needed to call protected back-end services.

After receiving a request to update the basket, the aggregator service calls the Catalog API to get the item details. The Basket controller uses an injected `ICatalogService` object to make that call and communicate with the Catalog API. The original implementation of the interface used gRPC to make the call. The updated implementation uses Dapr service invocation with `HttpClient` support:

```
public class CatalogService : ICatalogService
{
    private readonly HttpClient _httpClient;

    public CatalogService(HttpClient httpClient)
    {
        _httpClient = httpClient ?? throw new ArgumentNullException(nameof(httpClient));
    }

    public Task<IEnumerable<CatalogItem>> GetCatalogItemsAsync(IEnumerable<int> ids)
    {
        var requestUri = $"api/v1/catalog/items?ids={string.Join(",", ids)}";

        return _httpClient.GetFromJsonAsync<IEnumerable<CatalogItem>>(requestUri);
    }

    // ...
}
```

Notice how no Dapr specific code is required to make the service invocation call. All communication is done using the standard `HttpClient` object.

The Dapr `HttpClient` is injected into the `CatalogService` class in the `Startup.ConfigureServices` method:

```
services.AddSingleton<ICatalogService, CatalogService>(
    _ => new CatalogService(DaprClient.CreateInvokeHttpClient("catalog-api")));
```

The other call made by the aggregator service is to the Basket API. It only allows authorized requests. The access token is passed along in an `Authorization` request header to ensure the call succeeds:

```
public class BasketService : IBasketService
{
    public Task UpdateAsync(BasketData currentBasket, string accessToken)
    {
        var request = new HttpRequestMessage(HttpMethod.Post, "api/v1/basket")
        {
            Content = JsonContent.Create(currentBasket)
        };
        request.Headers.Authorization = new AuthenticationHeaderValue(accessToken);

        var response = await _httpClient.SendAsync(request);
        response.EnsureSuccessStatusCode();
    }

    // ...
}
```

In this example too, only standard `HttpClient` functionality is used to call the service. This allows developers who are already familiar with `HttpClient` to reuse their existing skills. It even enables existing `HttpClient` code to use Dapr service invocation without making any changes.

Summary

In this chapter, you learned about the service invocation building block. You saw how to invoke remote methods both by making direct HTTP calls to the Dapr sidecar, and by using the Dapr .NET SDK.

The Dapr .NET SDK provides multiple ways to invoke remote methods. HttpClient support is great for developers wanting to reuse existing skills and is compatible with many existing frameworks and libraries. DaprClient offers support for directly using the Dapr service invocation API using either HTTP or gRPC semantics.

The eShopOnDapr reference architecture shows how the original eShopOnContainers solution is modernized by using Dapr service invocation. Adding Dapr to eShop provides benefits such as automatic retries, message encryption using mTLS, and improved observability.

References

- [Dapr service invocation building block](#)
- [Monitoring distributed cloud-native applications](#)

The Dapr publish & subscribe building block

The [Publish-Subscribe pattern](#) (often referred to as “pub/sub”) is a well-known and widely used messaging pattern. Architects commonly embrace it in distributed applications. However, the plumbing to implement it can be complex. There are often subtle feature differences across different messaging products. Dapr offers a building block that significantly simplifies implementing pub/sub functionality.

What it solves

The primary advantage of the Publish-Subscribe pattern is **loose coupling**, sometimes referred to as [temporal decoupling](#). The pattern decouples services that send messages (the **publishers**) from services that consume messages (the **subscribers**). Both publishers and subscribers are unaware of each other - both are dependent on a centralized **message broker** that distributes the messages.

Figure 7-1 shows the high-level architecture of the pub/sub pattern.

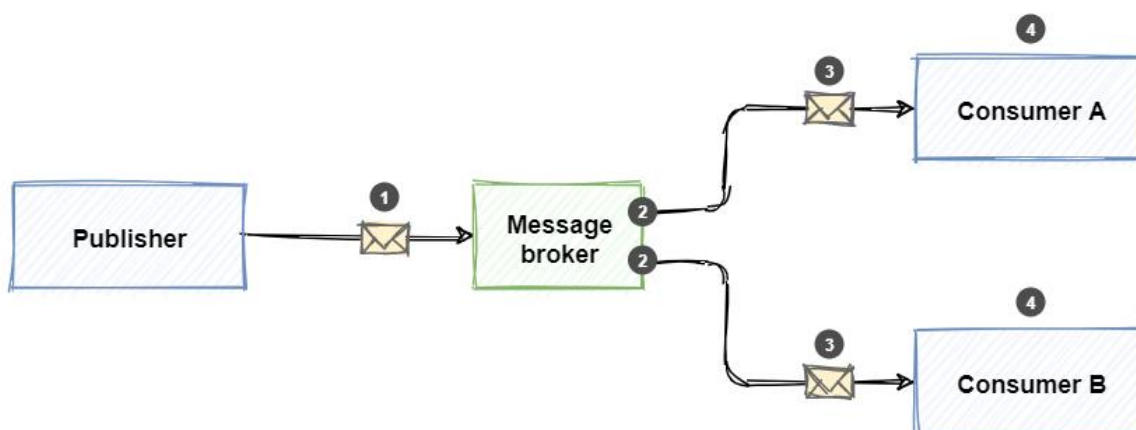


Figure 7-1. The pub/sub pattern.

From the previous figure, note the steps of the pattern:

1. Publishers send messages to the message broker.
2. Subscribers bind to a subscription on the message broker.

3. The message broker forwards a copy of the message to interested subscriptions.
4. Subscribers consume messages from their subscriptions.

Most message brokers encapsulate a queueing mechanism that can persist messages once received. With it, the message broker guarantees **durability** by storing the message. Subscribers don't need to be immediately available or even online when a publisher sends a message. Once available, the subscriber receives and processes the message. Dapr guarantees **At-Least-Once** semantics for message delivery. Once a message is published, it will be delivered at least once to any interested subscriber.

If your service can only process a message once, you'll need to provide an [idempotency check](#) to ensure that the same message is not processed multiple times. While such logic can be coded, some message brokers, such as Azure Service Bus, provide built-in *duplicate detection* messaging capabilities.

There are several message broker products available - both commercially and open-source. Each has advantages and drawbacks. Your job is to match your system requirements to the appropriate broker. Once selected, it's a best practice to decouple your application from message broker plumbing. You achieve this functionality by wrapping the broker inside an *abstraction*. The abstraction encapsulates the message plumbing and exposes generic pub/sub operations to your code. Your code communicates with the abstraction, not the actual message broker. While a wise decision, you'll have to write and maintain the abstraction and its underlying implementation. This approach requires custom code that can be complex, repetitive, and error-prone.

The Dapr publish & subscribe building block provides the messaging abstraction and implementation out-of-the-box. The custom code you would have had to write is prebuilt and encapsulated inside the Dapr building block. You bind to it and consume it. Instead of writing messaging plumbing code, you and your team focus on creating business functionality that adds value to your customers.

How it works

The Dapr publish & subscribe building block provides a platform-agnostic API framework to send and receive messages. Your services publish messages to a named [topic](#). Your services subscribe to a topic to consume messages.

The service calls the pub/sub API on the Dapr sidecar. The sidecar then makes calls into a pre-defined Dapr pub/sub component that encapsulates a specific message broker product. Figure 7-2 shows the Dapr pub/sub messaging stack.

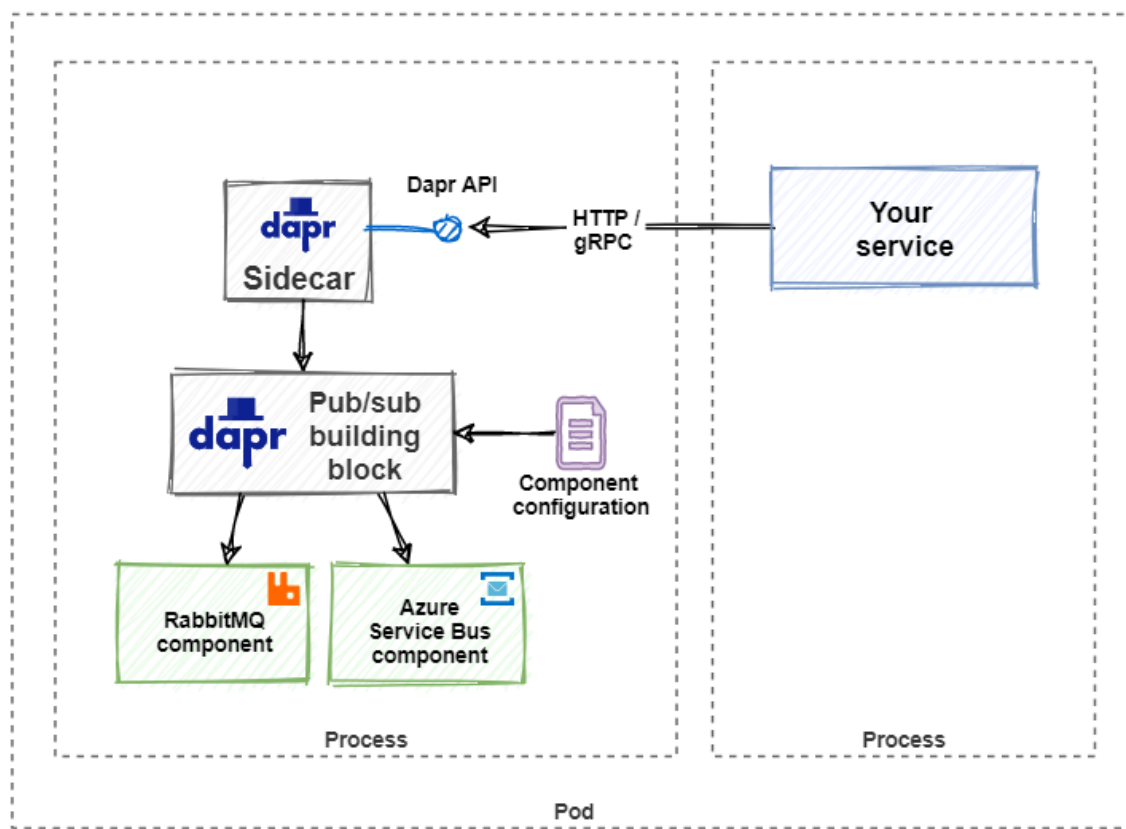


Figure 7-2. The Dapr pub/sub stack.

The Dapr publish & subscribe building block can be invoked in many ways.

At the lowest level, any programming platform can invoke the building block over HTTP or gRPC using the **Dapr native API**. To publish a message, you make the following API call:

```
http://localhost:<dapr-port>/v1.0/publish/<pub-sub-name>/<topic>
```

There are several Dapr specific URL segments in the above call:

- <dapr-port> provides the port number upon which the Dapr sidecar is listening.
- <pub-sub-name> provides the name of the selected Dapr pub/sub component.
- <topic> provides the name of the topic to which the message is published.

Using the *curl* command-line tool to publish a message, you can try it out:

```
curl -X POST http://localhost:3500/v1.0/publish/pubsub/newOrder \
-H "Content-Type: application/json" \
-d '{ "orderId": "1234", "productId": "5678", "amount": 2 }'
```

You receive messages by subscribing to a topic. At startup, the Dapr runtime will call the application on a well-known endpoint to identify and create the required subscriptions:

```
http://localhost:<appPort>/dapr/subscribe
```

- <appPort> informs the Dapr sidecar of the port upon which the application is listening.

You can implement this endpoint yourself. But Dapr provides more intuitive ways of implementing it. We'll address this functionality later in this chapter.

The response from the call contains a list of topics to which the applications will subscribe. Each includes an endpoint to call when the topic receives a message. Here's an example of a response:

```
[
  {
    "pubsubname": "pubsub",
    "topic": "newOrder",
    "route": "/orders"
  },
  {
    "pubsubname": "pubsub",
    "topic": "newProduct",
    "route": "/productCatalog/products"
  }
]
```

In the JSON response, you can see the application wants to subscribe to topics `newOrder` and `newProduct`. It registers the endpoints `/orders` and `/productCatalog/products` for each, respectively. For both subscriptions, the application is binding to the Dapr component named `pubsub`.

Figure 7-3 presents the flow of the example.

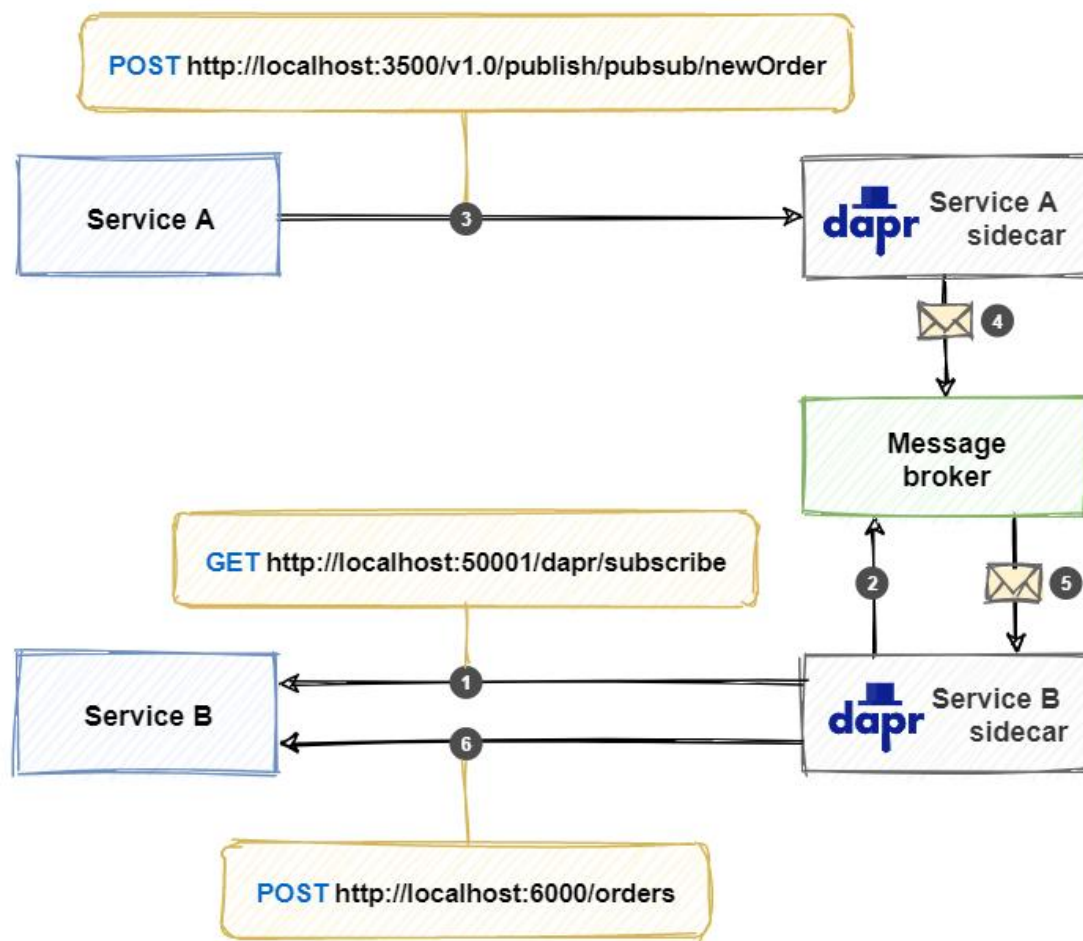


Figure 7-3. pub/sub flow with Dapr.

From the previous figure, note the flow:

1. The Dapr sidecar for Service B calls the `/dapr/subscribe` endpoint from Service B (the consumer). The service responds with the subscriptions it wants to create.
2. The Dapr sidecar for Service B creates the requested subscriptions on the message broker.
3. Service A publishes a message at the `/v1.0/publish/<pub-sub-name>/<topic>` endpoint on the Dapr Service A sidecar.
4. The Service A sidecar publishes the message to the message broker.
5. The message broker sends a copy of the message to the Service B sidecar.
6. The Service B sidecar calls the endpoint corresponding to the subscription (in this case `/orders`) on Service B. The service responds with an HTTP status-code 200 OK so the sidecar will consider the message as being handled successfully.

In the example, the message is handled successfully. But if something goes wrong while Service B is handling the request, it can use the response to specify what needs to happen with the message. When it returns an HTTP status-code 404, an error is logged and the message is dropped. With any other status-code than 200 or 404, a warning is logged and the message is retried. Alternatively,

Service B can explicitly specify what needs to happen with the message by including a JSON payload in the body of the response:

```
{
  "status": "<status>"
}
```

The following table shows the available status values:

Status	Action
SUCCESS	The message is considered as processed successfully and dropped.
RETRY	The message is retried.
DROP	A warning is logged and the message is dropped.
Any other status	The message is retried.

Competing consumers

When scaling out an application that subscribes to a topic, you have to deal with competing consumers. Only one application instance should handle a message sent to the topic. Luckily, Dapr handles that problem. When multiple instances of a service with the same application-id subscribe to a topic, Dapr delivers each message to only one of them.

SDKs

Making HTTP calls to the native Dapr APIs is time-consuming and abstract. Your calls are crafted at the HTTP level, and you'll need to handle plumbing concerns such as serialization and HTTP response codes. Fortunately, there's a more intuitive way. Dapr provides several language-specific SDKs for popular development platforms. At the time of this writing, Go, Node.js, Python, .NET, Java, and JavaScript are available.

Use the Dapr .NET SDK

For .NET Developers, the [Dapr .NET SDK](#) provides a more productive way of working with Dapr. The SDK exposes a `DaprClient` class through which you can directly invoke Dapr functionality. It's intuitive and easy to use.

To publish a message, the `DaprClient` exposes a `PublishEventAsync` method.

```
var data = new OrderData
{
    orderId = "123456",
    productId = "67890",
    amount = 2
}

var daprClient = new DaprClientBuilder().Build();

await daprClient.PublishEventAsync<OrderData>("pubsub", "newOrder", data);
```

- The first argument pubsub is the name of the Dapr component that provides the message broker implementation. We'll address components later in this chapter.
- The second argument newOrder provides the name of the topic to send the message to.
- The third argument is the payload of the message.
- You can specify the .NET type of the message using the generic type parameter of the method.

To receive messages, you bind an endpoint to a subscription for a registered topic. The ASP.NET Core library for Dapr makes this trivial. Assume, for example, that you have an existing ASP.NET WebAPI action method entitled CreateOrder:

```
[HttpPost("/orders")]
public async Task<ActionResult> CreateOrder(Order order)
```

You must add a reference to the [Dapr.AspNetCore](#) NuGet package in your project to consume the Dapr ASP.NET Core integration.

To bind this action method to a topic, you decorate it with the Topic attribute:

```
[Topic("pubsub", "newOrder")]
[HttpPost("/orders")]
public async Task<ActionResult> CreateOrder(Order order)
```

You specify two key elements with this attribute:

- The Dapr pub/sub component to target (in this case pubsub).
- The topic to subscribe to (in this case newOrder).

Dapr then invokes that action method as it receives messages for that topic.

You'll also need to enable ASP.NET Core to use Dapr. The Dapr .NET SDK provides several extension methods that can be invoked in the Startup class.

In the ConfigureServices method, you must add the following extension method:

```
public void ConfigureServices(IServiceCollection services)
{
    // ...
    services.AddControllers().AddDapr();
}
```

Appending the AddDapr extension-method to the AddControllers extension-method registers the necessary services to integrate Dapr into the MVC pipeline. It also registers a DaprClient instance into the dependency injection container, which then can be injected anywhere into your service.

In the Configure method, you must add the following middleware components to enable Dapr:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // ...
    app.UseCloudEvents();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapSubscribeHandler();
    });
}
```

```
    // ...  
});  
}
```

The call to `UseCloudEvents` adds **CloudEvents** middleware into to the ASP.NET Core middleware pipeline. This middleware will unwrap requests that use the CloudEvents structured format, so the receiving method can read the event payload directly.

[CloudEvents](#) is a standardized messaging format, providing a common way to describe event information across platforms. Dapr embraces CloudEvents. For more information about CloudEvents, see the [cloudevents specification](#).

The call to `MapSubscribeHandler` in the endpoint routing configuration will add a Dapr subscribe endpoint to the application. This endpoint will respond to requests on `/dapr/subscribe`. When this endpoint is called, it will automatically find all WebAPI action methods decorated with the `Topic` attribute and instruct Dapr to create subscriptions for them.

Pub/sub components

Dapr [pub/sub components](#) handle the actual transport of the messages. Several are available. Each encapsulates a specific message broker product to implement the pub/sub functionality. At the time of writing, the following pub/sub components were available:

- Apache Kafka
- Azure Event Hubs
- Azure Service Bus
- AWS SNS/SQS
- GCP Pub/Sub
- Hazelcast
- MQTT
- NATS
- Pulsar
- RabbitMQ
- Redis Streams

Note

The Azure cloud stack has both messaging functionality (Azure Service Bus) and event streaming (Azure Event Hub) availability.

These components are created by the community in a [component-contrib repository on GitHub](#). You're encouraged to write your own Dapr component for a message broker that isn't yet supported.

Configure pub/sub components

Using a Dapr configuration file, you can specify the pub/sub component(s) to use. This configuration contains several fields. The `name` field specifies the pub/sub component that you want to use. When

sending or receiving a message, you need to specify this name (as you saw earlier in the `PublishEventAsync` method signature).

Below you see an example of a Dapr configuration file for configuring a RabbitMQ message broker component:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: pubsub-rq
spec:
  type: pubsub.rabbitmq
  version: v1
  metadata:
    - name: host
      value: "amqp://localhost:5672"
    - name: durable
      value: true
```

In this example, you can see that you can specify any message broker-specific configuration in the metadata block. In this case, RabbitMQ is configured to create durable queues. But the RabbitMQ component has more configuration options. Each of the components' configuration will have its own set of possible fields. You can read which fields are available in the documentation of each [pub/sub component](#).

Next to the programmatic way of subscribing to a topic from code, Dapr pub/sub also provides a declarative way of subscribing to a topic. This approach removes the Dapr dependency from the application code. Therefore, it also enables an existing application to subscribe to topics without any changes to the code. The following example shows a Dapr configuration file for configuring a subscription:

```
apiVersion: dapr.io/v1alpha1
kind: Subscription
metadata:
  name: newOrder-subscription
spec:
  pubsubname: pubsub
  topic: newOrder
  route: /orders
scopes:
  - ServiceB
  - ServiceC
```

You have to specify several elements with every subscription:

- The name of the Dapr pub/sub component you want to use (in this case `pubsub`).
- The name of the topic to subscribe to (in this case `newOrder`).
- The API operation that needs to be called for this topic (in this case `/orders`).
- The [scope](#) can specify which services can publish and subscribe to a topic.

Reference application: eShopOnDapr

The accompanying [eShopOnDapr](#) app provides an end-to-end reference architecture for constructing a microservices application implementing Dapr. eShopOnDapr is an evolution of the widely popular [eShopOnContainers](#) app, created several years ago. Both versions use the pub/sub pattern for communicating [integration events](#) across microservices. Integration events include:

- When a user checks-out a shopping basket.
- When a payment for an order has succeeded.
- When the grace-period of a purchase has expired.

Eventing in eShopOnContainers is based on the following `IEventBus` interface:

```
public interface IEventBus
{
    void Publish(IntegrationEvent integrationEvent);

    void Subscribe<T, THandler>()
        where TEvent : IntegrationEvent
        where THandler : IIntegrationEventHandler<T>;
}
```

Concrete implementations of this interface exist in eShopOnContainers for both RabbitMQ and Azure Service Bus. Each implementation included a great deal of custom plumbing code that was complex to understand and difficult to maintain.

The newer eShopOnDapr significantly simplifies pub/sub behavior by using Dapr. For example, the `IEventBus` interface was reduced to a single method:

```
public interface IEventBus
{
    Task PublishAsync(IntegrationEvent integrationEvent);
}
```

Publish events

In the updated eShopOnDapr, a single `DaprEventBus` implementation can support any Dapr-supported message broker. The following code block shows the simplified `Publish` method. Note how the `PublishAsync` method uses the Dapr client to publish an event:

```
public class DaprEventBus : IEventBus
{
    private const string PubSubName = "pubsub";

    private readonly DaprClient _daprClient;
    private readonly ILogger<DaprEventBus> _logger;

    public DaprEventBus(DaprClient daprClient, ILogger<DaprEventBus> logger)
    {
        _daprClient = daprClient ?? throw new ArgumentNullException(nameof(daprClient));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public async Task PublishAsync(IntegrationEvent integrationEvent)
```

```

{
    var topicName = integrationEvent.GetType().Name;

    // Dapr uses System.Text.Json which does not support serialization of a
    // polymorphic type hierarchy by default. Using object as the type
    // parameter causes all properties to be serialized.
    await _daprClient.PublishEventAsync<object>(PubSubName, topicName,
integrationEvent);
}
}

```

As you can see in the code snippet, the topic name is derived from event type's name. Because all eShop services use the IEventBus abstraction, retrofitting Dapr required *absolutely no change* to the mainline application code.

Important

The Dapr SDK uses System.Text.Json to serialize/deserialize messages. However, System.Text.Json doesn't serialize properties of derived classes by default. In the eShop code, an event is sometimes explicitly declared as an IntegrationEvent, the base class for integration events. This is done because the concrete event type is determined dynamically at run time based on business logic. As a result, the event is serialized using the type information of the base class and not the derived class. To force System.Text.Json to serialize all properties of the derived class in this case, the code uses object as the generic type parameter. For more information, see the [.NET documentation](#).

With Dapr, the infrastructure code is **dramatically simplified**. It doesn't need to distinguish between the different message brokers. Dapr provides this abstraction for you. And if needed, you can easily swap out message brokers or configure multiple message broker components.

Subscribe to events

The earlier eShopOnContainers app contains *SubscriptionManagers* to handle the subscription implementation for each message broker. Each manager contains complex message broker-specific code for handling subscription events. To receive events, each service has to explicitly register a handler for each event-type.

eShopOnDapr streamlines the plumbing for event subscriptions by using Dapr ASP.NET Core libraries. Each event is handled by an action method in the controller. A Topic attribute decorates the action method with the name of the corresponding topic to subscribe to. Here's a code snippet taken from the PaymentService:

```

[Route("api/v1/[controller]")]
[ApiController]
public class IntegrationEventController : ControllerBase
{
    private const string DAPR_PUBSUB_NAME = "pubsub";

    private readonly IServiceProvider _serviceProvider;

    public IntegrationEventController(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }
}

```

```

[HttpPost("OrderStatusChangedToValidated")]
[Topic(DAPR_PUBSUB_NAME, "OrderStatusChangedToValidatedIntegrationEvent")]
public async Task OrderStarted(OrderStatusChangedToValidatedIntegrationEvent
integrationEvent)
{
    var handler =
_serviceProvider.GetRequiredService<OrderStatusChangedToValidatedIntegrationEventHandler>()
;
    await handler.Handle(integrationEvent);
}
}

```

In the Topic attribute, the name of the .NET type of the event is used as the topic name. For handling the event, an event handler that already existed in the earlier eShopOnContainers code base is invoked. In the previous example, messages received from the OrderStatusChangedToValidatedIntegrationEvent topic invoke the existing OrderStatusChangedToValidatedIntegrationEventHandler event-handler. Because Dapr implements the underlying plumbing for subscriptions and message brokers, a large amount of original code became obsolete and was removed from the code-base. Much of this code was complex to understand and challenging to maintain.

Use pub/sub components

Within the eShopOnDapr repository, a deployment folder contains files for deploying the application using different deployment modes: Docker Compose and Kubernetes. A dapr folder exists within each of these folders that holds a components folder. This folder holds a file eshop-pubsub.yaml containing the configuration of the Dapr pub/sub component that the application will use for pub/sub behavior. As you saw in the earlier code snippets, the name of the pub/sub component used is pubsub. Here's the content of the eshop-pubsub.yaml file in the deployment/compose/dapr/components folder:

```

apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: pubsub
  namespace: default
spec:
  type: pubsub.nats
  version: v1
  metadata:
    - name: natsURL
      value: nats://demo.nats.io:4222

```

The preceding configuration specifies the desired [NATS message broker](#) for this example. To change message brokers, you need only to configure a different message broker, such as RabbitMQ or Azure Service Bus and update the yaml file. With Dapr, there are no changes to your mainline service code when switching message brokers.

Finally, you might ask, "Why would I need multiple message brokers in an application?". Many times a system will handle workloads with different characteristics. One event may occur 10 times a day, but another event occurs 5,000 times per second. You may benefit by partitioning messaging traffic to

different message brokers. With Dapr, you can add multiple pub/sub component configurations, each with a different name.

Summary

The pub/sub pattern helps you decouple services in a distributed application. The Dapr publish & subscribe building block simplifies implementing this behavior in your application.

Through Dapr pub/sub, you can publish messages to a specific *topic*. As well, the building block will query your service to determine which topic(s) to subscribe to.

You can use Dapr pub/sub natively over HTTP or by using one of the language-specific SDKs, such as the .NET SDK for Dapr. The .NET SDK tightly integrates with the ASP.NET core platform.

With Dapr, you can plug a supported message broker product into your application. You can then swap message brokers without requiring code changes to your application.

The Dapr bindings building block

Cloud-based *serverless* offerings, such as Azure Functions and AWS Lambda, have gained wide adoption across the distributed architecture space. Among many benefits, they enable a microservice to *handle events from or invoke events in* an external system - abstracting away the underlying complexity and plumbing concerns. External resources are many: They include datastores, message systems, and web resources, across different platforms and vendors. The [Dapr bindings building block](#) brings these same resource binding capabilities to the doorstep of your Dapr applications.

What it solves

Dapr resource bindings enable your services to integrate business operations across external resources outside of the immediate application. An event from an external system could trigger an operation in your service passing in contextual information. Your service could then expand the operation by triggering an event in another external system, passing in contextual payload information. Your service communicates without coupling or awareness of the external resource. The plumbing is encapsulated inside pre-defined Dapr components. The Dapr component to use can be easily swapped at runtime without code changes.

Consider, for example, a Twitter account that triggers an event whenever a user tweets a keyword. Your service exposes an event handler that receives and processes the tweet. Once complete, your service triggers an event that invokes an external Twilio service. Twilio sends an SMS message that includes the tweet. Figure 8-1 show the conceptual architecture of this operation.

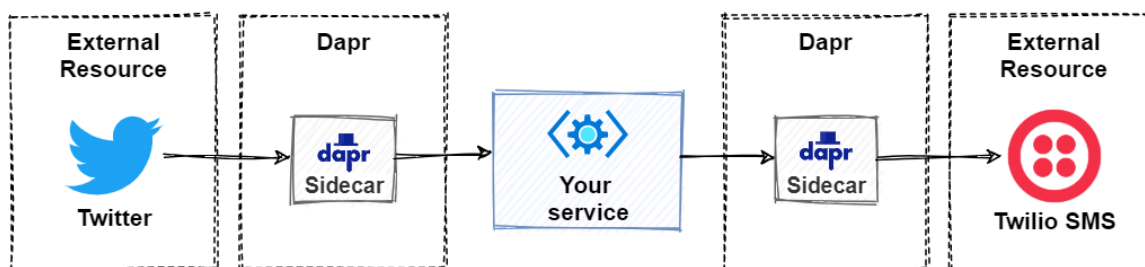


Figure 8-1. Conceptual architecture of a Dapr resource binding.

At first glance, resource binding behavior may appear similar to the [Publish/Subscribe pattern](#) described earlier in this book. While they share similarities, there are differences. Publish/subscribe

focuses on asynchronous communication between Dapr services. Resource binding has a much wider scope. It focuses on system interoperability across software platforms. Exchanging information between disparate applications, datastores, and services outside your microservice application.

How it works

Dapr resource binding starts with a component configuration file. This YAML file describes the type of resource to which you'll bind along with its configuration settings. Once configured, your service can receive events from the resource or trigger events on it.

Note

Binding configurations are presented in detail later in the *Components* section.

Input bindings

Input bindings trigger your code with incoming events from external resources. To receive events and data, you register a public endpoint from your service that becomes the *event handler*. Figure 8-2 shows the flow:

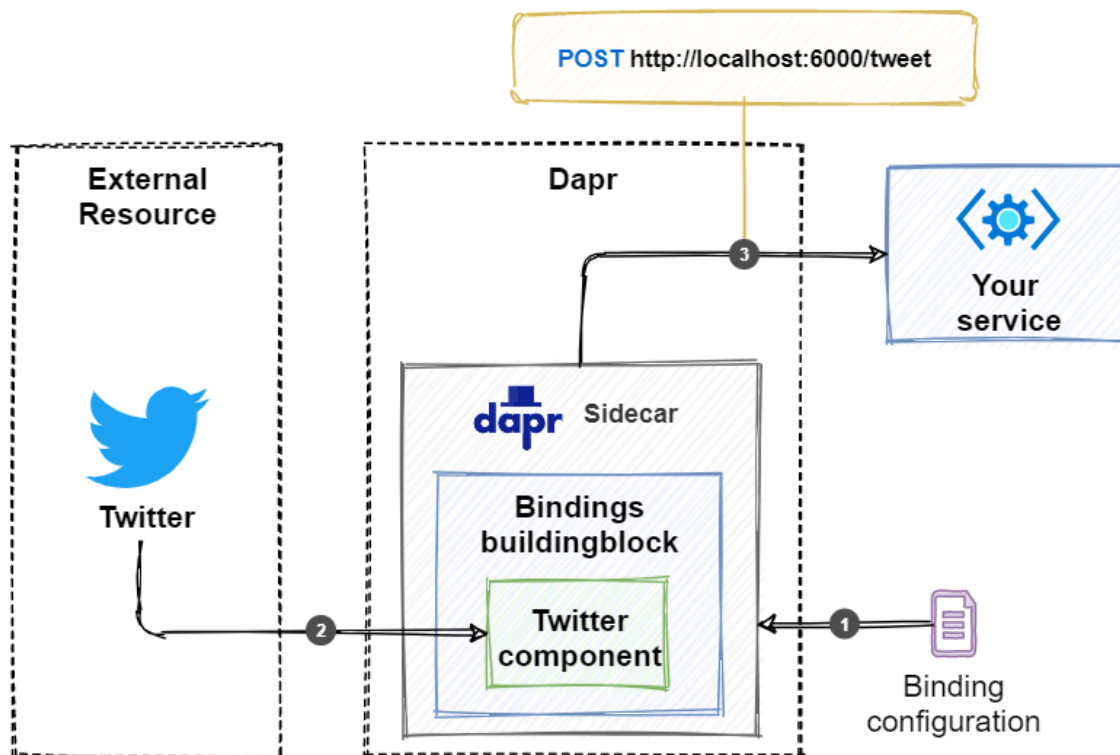


Figure 8-2. Dapr input binding flow.

Figure 8.2 describes the steps for receiving events from an external Twitter account:

1. The Dapr sidecar reads the binding configuration file and subscribes to the event specified for the external resource. In the example, the event source is a Twitter account.
2. When a matching Tweet is published on Twitter, the binding component running in the Dapr sidecar picks it up and triggers an event.
3. The Dapr sidecar invokes the endpoint (that is, event handler) configured for the binding. In the example, the service listens for an HTTP POST on the /tweet endpoint on port 6000. Because it's an HTTP POST operation, the JSON payload for the event is passed in the request body.
4. After handling the event, the service returns an HTTP status code 200 OK.

The following ASP.NET Core controller provides an example of handling an event triggered by the Twitter binding:

```
[ApiController]
public class SomeController : ControllerBase
{
    public class TwitterTweet
    {
        [JsonPropertyName("id_str")]
        public string ID {get; set; }

        [JsonPropertyName("text")]
        public string Text {get; set; }
    }

    [HttpPost("/tweet")]
    public ActionResult Post(TwitterTweet tweet)
    {
        // Handle tweet
        Console.WriteLine("Tweet received: {0}: {1}", tweet.ID, tweet.Text);

        // ...

        // Acknowledge message
        return Ok();
    }
}
```

If the operation should error, you would return the appropriate 400 or 500 level HTTP status code. For bindings that feature *at-least-once-delivery* guarantees, the Dapr sidecar will retry the trigger. Check out [Dapr documentation for resource bindings][1] to see whether they offer *at-least-once* or *exactly-once* delivery guarantees.

Output bindings

Dapr also includes *output binding* capabilities. They enable your service to trigger an event that invokes an external resource. Again, you start by configuring a binding configuration YAML file that describes the output binding. Once in place, you trigger an event that invokes the bindings API on the Dapr sidecar of your application. Figure 8-3 shows the flow of an output binding:

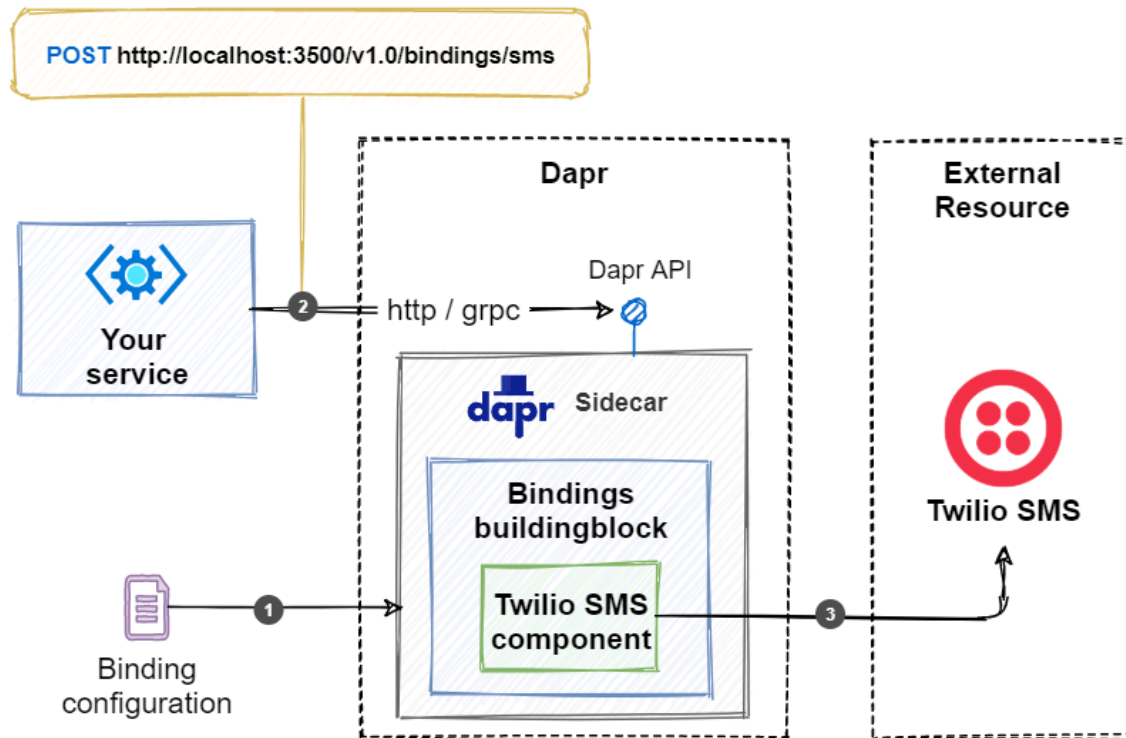


Figure 8-3. Dapr output binding flow.

1. The Dapr sidecar reads the binding configuration file with the information on how to connect to the external resource. In the example, the external resource is a Twilio SMS account.
2. Your application invokes the `/v1.0/bindings/sms` endpoint on the Dapr sidecar. In this case, it uses an HTTP POST to invoke the API. It's also possible to use gRPC.
3. The binding component running in the Dapr sidecar calls the external messaging system to send the message. The message will contain the payload passed in the POST request.

As an example, you can invoke an output binding by invoking the Dapr API using curl:

```
curl -X POST http://localhost:3500/v1.0/bindings/sms \
  -H "Content-Type: application/json" \
  -d '{
    "data": "Welcome to this awesome service",
    "metadata": {
      "toNumber": "555-3277"
    },
    "operation": "create"
  }'
```

Note that the HTTP port is the same as used by the Dapr sidecar (in this case, the default Dapr HTTP port 3500).

The structure of the payload (that is, message sent) will vary per binding. In the example above, the payload contains a data element with a message. Bindings to other types of external resources can be different, especially for the metadata that is sent. Each payload must also contain an operation field,

that defines the operation the binding will execute. The above example specifies a create operation that creates the SMS message. Common operations include:

- create
- get
- delete
- list

It's up to the author of the binding which operations the binding supports. The documentation for each binding describes the available operations and how to invoke them.

Using the Dapr .NET SDK

The Dapr .NET SDK provides language-specific support for .NET Core developers. In the following example, the call to the `HttpClient.PostAsync()` is replaced with the `DaprClient.InvokeBindingAsync()` method. This specialized method simplifies invoking a configured output binding:

```
private async Task SendSMSAsync([FromServices] DaprClient daprClient)
{
    var message = "Welcome to this awesome service";
    var metadata = new Dictionary<string, string>
    {
        { "toNumber", "555-3277" }
    };
    await daprClient.InvokeBindingAsync("sms", "create", message, metadata);
}
```

The method expects the metadata and message values.

When used to invoke a binding, the `DaprClient` uses gRPC to call the Dapr API on the Dapr sidecar.

Binding components

Under the hood, resource bindings are implemented with Dapr binding components. They're contributed by the community and written in Go. If you need to integrate with an external resource for which no Dapr binding exists yet, you can create it yourself. Check out the [Dapr components-contrib repo](#) to see how you can contribute a binding.

Note

Dapr and all of its components are written in the [Golang](#) (Go) language. Go is considered a modern, cloud-native programming platform.

You configure bindings using a YAML configuration file. Here's an example configuration for the Twitter binding:

```
apiVersion: daprio.io/v1alpha1
kind: Component
metadata:
  name: twitter-mention
```

```

namespace: default
spec:
  type: bindings.twitter
  version: v1
  metadata:
    - name: consumerKey
      value: "*****" # twitter api consumer key, required
    - name: consumerSecret
      value: "*****" # twitter api consumer secret, required
    - name: accessToken
      value: "*****" # twitter api access token, required
    - name: accessSecret
      value: "*****" # twitter api access secret, required
    - name: query
      value: "dapr" # your search query, required

```

Each binding configuration contains a general metadata element with a name and namespace field. Dapr will determine the endpoint to invoke your service based upon the configured name field. In the above example, Dapr will invoke the method annotated with `/twitter-mention` in your service when an event occurs.

In the `spec` element, you specify the type of the binding along with binding specific metadata. The example specifies credentials for accessing a Twitter account using its API. The metadata can differ between input and output bindings. For example, to use Twitter as an input binding, you need to specify the text to search for in tweets using the `query` field. Every time a matching tweet is sent, the Dapr sidecar will invoke the `/twitter-mention` endpoint on the service. It will also deliver the contents of the tweet.

A binding can be configured for input, output, or both. Interestingly, the binding doesn't explicitly specify input or output configuration. Instead, the direction is inferred by the usage of the binding along with configuration values.

The [Dapr documentation for resource bindings][1] provides a complete list of the available bindings and their specific configuration settings.

Cron binding

Pay close attention to Dapr's Cron binding. It doesn't subscribe to events from an external system. Instead, this binding uses a configurable interval schedule to trigger your application. The binding provides a simple way to implement a background worker to wake up and do some work at a regular interval, without the need to implement an endless loop with a configurable delay. Here's an example of a Cron binding configuration:

```

apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: checkOrderBacklog
  namespace: default
spec:
  type: bindings.cron
  version: v1
  metadata:
    - name: schedule
      value: "@every 30m"

```

In this example, Dapr triggers a service by invoking the `/checkOrderBacklog` endpoint every 30 minutes. There are several patterns available for specifying the schedule value. For more information, see the [Cron binding documentation](#).

Reference application: eShopOnDapr

The accompanying eShopOnDapr reference application implements an output binding example. It triggers the Dapr [SendGrid](#) binding to send a user an email when a new order is placed. You can find this binding in the `eshop-email.yaml` file in the `components` folder:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: sendmail
  namespace: eshop
spec:
  type: bindings.twilio.sendgrid
  version: v1
  metadata:
    - name: apiKey
      secretKeyRef:
        name: sendGridAPIKey
  auth:
    secretStore: eshop-secretstore
```

This configuration uses the [Twilio SendGrid](#) binding component. Note how the API key for connecting to the service consumes a Dapr secret reference. This approach keeps secrets outside of the configuration file. Read the [secrets building block chapter](#) to learn more about Dapr secrets.

The binding configuration specifies a binding component that can be invoked using the `/sendmail` endpoint on the Dapr sidecar. Here's a code snippet in which an email is sent whenever an order is started:

```
public Task Handle(OrderStartedDomainEvent notification, CancellationToken
cancellation_token)
{
    var string message = CreateEmailBody(notification);
    var metadata = new Dictionary<string, string>
    {
        {"emailFrom", "eShopOn@dapr.io"},
        {"emailTo", notification.UserName},
        {"subject", $"Your eShopOnDapr order #{notification.Order.Id}"}
    };
    return _daprClient.InvokeBindingAsync("sendmail", "create", message, metadata,
cancellation_token);
}
```

As you can see in this example, `message` contains the message body. The `CreateEmailBody` method simply formats a string with the body text. The `metadata` specifies the email sender, recipient, and the subject for the email message. If these values are static, they can also be included in the `metadata` fields in the configuration file. The name of the binding to invoke is `sendmail` and the operation is `create`.

Summary

Dapr resource bindings enable you to integrate with different external resources and systems without taking dependencies on their libraries or SDKs. These external systems don't necessarily have to be messaging systems like a service bus or message broker. Bindings also exist for datastores and web resources like Twitter or SendGrid.

Input bindings (or triggers) react to events occurring in an external system. They invoke the public HTTP endpoints pre-configured in your application. Dapr uses the name of the binding in the configuration to determine the endpoint to call in your application.

Output bindings will send messages to an external system. You trigger an output binding by doing an HTTP POST on the `/v1.0/bindings/<binding-name>` endpoint on the Dapr sidecar. You can also use gRPC to invoke the binding. The .NET SDK offers a `InvokeBindingAsync` method to invoke Dapr bindings using gRPC.

You implement a binding with a Dapr component. These components are contributed by the community. Each binding component's configuration has metadata that is specific for the external system it abstracts. Also, the commands it supports and the structure of the payload will differ per binding component.

References

- [Dapr documentation for resource bindings](#)

The Dapr observability building block

Modern distributed systems are complex. You start with small, loosely coupled, independently deployable services. These services cross process and server boundaries. They then consume different kinds of infrastructure backing services (databases, message brokers, key vaults). Finally, these disparate pieces compose together to form an application.

With so many separate, moving parts, how do you make sense of what is going on? Unfortunately, legacy monitoring approaches from the past aren't enough. Instead, the system must be **observable** from end-to-end. Modern [observability](#) practices provide visibility and insight into the health of the application at all times. They enable you to infer the internal state by observing the output. Observability is mandatory for monitoring and troubleshooting distributed applications.

The system information used to gain observability is referred to as **telemetry**. It can be divided into four broad categories:

1. **Distributed tracing** provides insight into the traffic between services and services involved in distributed transactions.
2. **Metrics** provides insight into the performance of a service and its resource consumption.
3. **Logging** provides insight into how the code is executing and if errors have occurred.
4. **Health** endpoints provide insight into the availability of a service.

The depth of telemetry is determined by the observability features of an application platform. Consider the Azure cloud. It provides a rich telemetry experience that includes all of the telemetry categories. Without any configuration, most Azure IaaS and PaaS services propagate and publish telemetry to the [Azure Application Insights](#) service. Application Insights presents system logging, tracing, and problem areas with highly visual dashboards. It can even render a diagram showing the dependencies between services based on their communication.

However, what if an application can't use Azure PaaS and IaaS resources? Is it still possible to take advantage of the rich telemetry experience of Application Insights? The answer is yes. A non-Azure application can import libraries, add configuration, and instrument code to emit telemetry to Azure Application Insights. However, this approach **tightly couples** the application to Application Insights. Moving the app to a different monitoring platform could involve expensive refactoring. Wouldn't it be great to avoid tight coupling and consume observability outside of the code?

With Dapr, you can. Let's look at how Dapr can add observability to our distributed applications.

What it solves

The Dapr observability building block decouples observability from the application. It automatically captures traffic generated by Dapr sidecars and Dapr system services that make up the Dapr control plane. The block correlates traffic from a single operation that spans multiple services. It also exposes performance metrics, resource utilization, and the health of the system. Telemetry is published in open-standard formats enabling information to be fed into your monitoring back end of choice. There, the information can be visualized, queried, and analyzed.

As Dapr abstracts away the plumbing, the application is unaware of how observability is implemented. There's no need to reference libraries or implement custom instrumentation code. Dapr allows the developer to focus on building business logic and not observability plumbing. Observability is configured at the Dapr level and is consistent across services, even when created by different teams, and built with different technology stacks.

How it works

Dapr's [sidecar architecture](#) enables built-in observability features. As services communicate, Dapr sidecars intercept the traffic and extract tracing, metrics, and logging information. Telemetry is published in an open standards format. By default, Dapr supports [OpenTelemetry](#) and [Zipkin](#).

Dapr provides [collectors](#) that can publish telemetry to different back-end monitoring tools. These tools present Dapr telemetry for analysis and querying. Figure 9-1 shows the Dapr observability architecture:

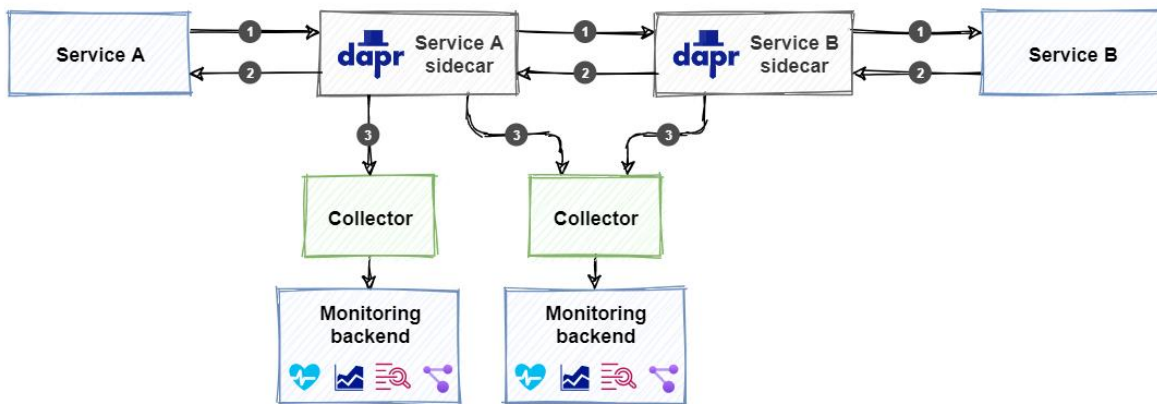


Figure 9-1. Dapr observability architecture.

1. Service A calls an operation on Service B. The call is routed from a Dapr sidecar for Service A to a sidecar for Service B.
2. When Service B completes the operation, a response is sent back to Service A through the Dapr sidecars. They gather and publish all available telemetry for every request and response.
3. The configured collector ingests the telemetry and sends it to the monitoring back end.

As a developer, keep in mind that adding observability is different from configuring other Dapr building blocks, like pub/sub or state management. Instead of referencing a building block, you add a collector and a monitoring back end. Figure 9-1 shows it's possible to configure multiple collectors that integrate with different monitoring back ends.

At the beginning of this chapter, four categories of telemetry were identified. The following sections will provide detail for each category. They'll include instruction on how to configure collectors that integrate with popular monitoring back ends.

Distributed tracing

Distributed tracing provides insight into the traffic that flows across services in a distributed application. The log of exchanged request and response messages is an invaluable source of information for troubleshooting issues. The hard part is *correlating messages* that originate from the same operation.

Dapr uses the [W3C Trace Context](#) to correlate related messages. It injects the same context information into requests and responses that form a unique operation. Figure 9-2 shows how correlation works:

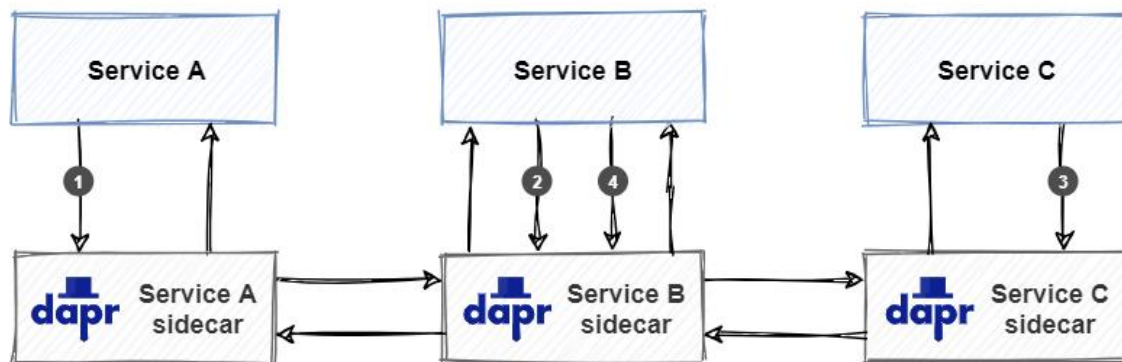


Figure 9-2. W3C Trace Context example.

1. Service A invokes an operation on Service B. As Service A starts the call, Dapr creates a unique trace context and injects it into the request.
2. Service B receives the request and invokes an operation on Service C. Dapr detects that the incoming request contains a trace context and propagates it by injecting it into the outgoing request to Service C.
3. Service C receives the request and handles it. Dapr detects that the incoming request contains a trace context and propagates it by injecting it into the outgoing response back to Service B.
4. Service B receives the response and handles it. It then creates a new response and propagates the trace context by injecting it into the outgoing response back to Service A.

A set of requests and responses that belong together is called a *trace*. Figure 9-3 shows a trace:

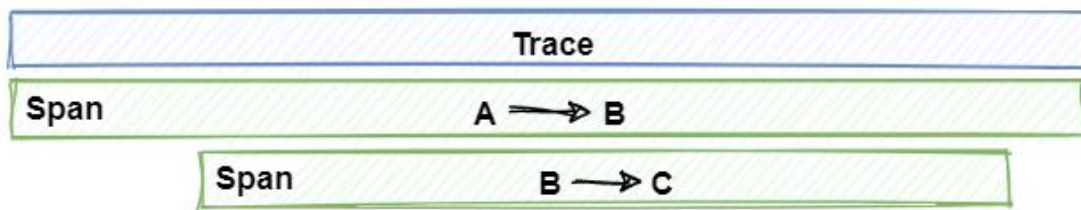


Figure 9-3. Traces and spans.

In the figure, note how the trace represents a unique application transaction that takes place across many services. A trace is a collection of *spans*. Each span represents a single operation or unit of work done within the trace. Spans are the requests and responses that are sent between services that implement the unique transaction.

The next sections discuss how to inspect tracing telemetry by publishing it to a monitoring back end.

Use a Zipkin monitoring back end

[Zipkin](#) is an open-source distributed tracing system. It can ingest and visualize telemetry data. Dapr offers default support for Zipkin. The following example demonstrates how to configure Zipkin to visualize Dapr telemetry.

Enable and configure tracing

To start, tracing must be enabled for the Dapr runtime using a Dapr configuration file. Here's an example of a configuration file named `tracing-config.yaml`:

```
apiVersion: daprio.io/v1alpha1
kind: Configuration
metadata:
  name: tracing-config
  namespace: default
spec:
  tracing:
    samplingRate: "1"
    zipkin:
      endpointAddress: "http://zipkin.default.svc.cluster.local:9411/api/v2/spans"
```

The `samplingRate` attribute specifies the interval used for publishing traces. The value must be between 0 (tracing disabled) and 1 (every trace is published). With a value of 0.5, for example, every other trace is published, significantly reducing published traffic. The `endpointAddress` points to an endpoint on a Zipkin server running in a Kubernetes cluster. The default port for Zipkin is 9411. The configuration must be applied to the Kubernetes cluster using the Kubernetes CLI:

```
kubectl apply -f tracing-config.yaml
```


Install the Zipkin server

When installing Dapr in self-hosted mode, a Zipkin server is automatically installed and tracing is enabled in the default configuration file located in `$HOME/.dapr/config.yaml` or `%USERPROFILE%\dapr\config.yaml` on Windows.

When installing Dapr on a Kubernetes cluster though, Zipkin isn't added by default. The following Kubernetes manifest file named `zipkin.yaml`, deploys a standard Zipkin server to the cluster:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: zipkin
  namespace: eshop
  labels:
    service: zipkin
spec:
  replicas: 1
  selector:
    matchLabels:
      service: zipkin
  template:
    metadata:
      labels:
        service: zipkin
    spec:
      containers:
        - name: zipkin
          image: openzipkin/zipkin-slim
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 9411
              protocol: TCP

---

kind: Service
apiVersion: v1
metadata:
  name: zipkin
  namespace: eshop
  labels:
    service: zipkin
spec:
  type: NodePort
  ports:
    - port: 9411
      targetPort: 9411
      nodePort: 32411
      protocol: TCP
      name: zipkin
  selector:
    service: zipkin
```

The deployment uses the standard `openzipkin/zipkin-slim` container image. The Zipkin service exposes the Zipkin web front end, which you can use to view the telemetry on port 32411. Use the

Kubernetes CLI to apply the Zipkin manifest file to the Kubernetes cluster and deploy the Zipkin server:

```
kubectl apply -f zipkin.yaml
```

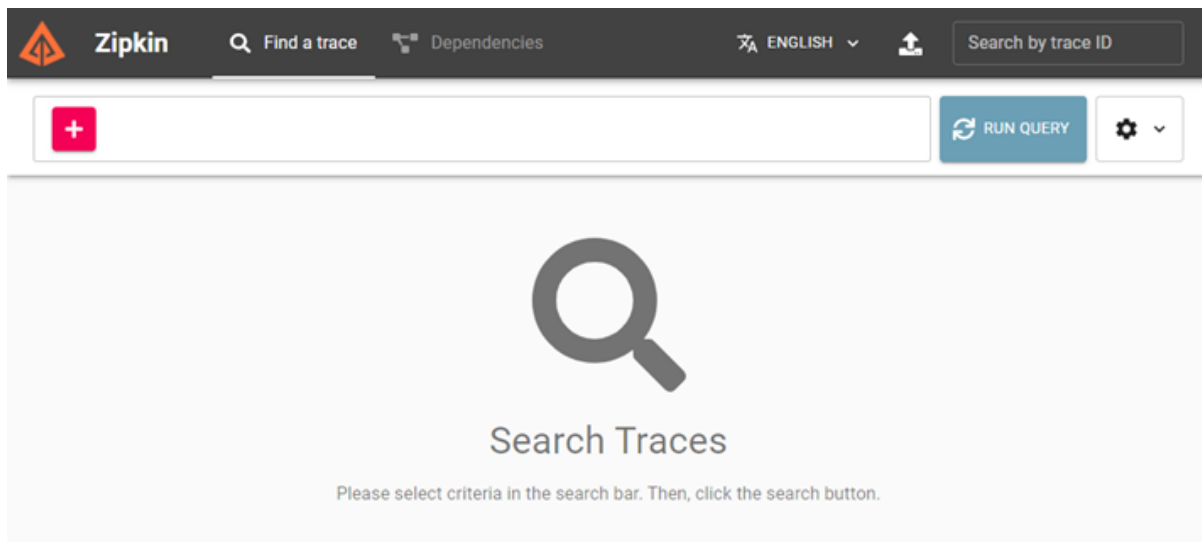
Configure the services to use the tracing configuration

Now everything is set up correctly to start publishing telemetry. Every Dapr sidecar that is deployed as part of the application must be instructed to emit telemetry when started. To do that, add a `dapr.io/config` annotation that references the tracing-config configuration to the deployment of each service. Here's an example of the eShop ordering API service's manifest file containing the annotation:

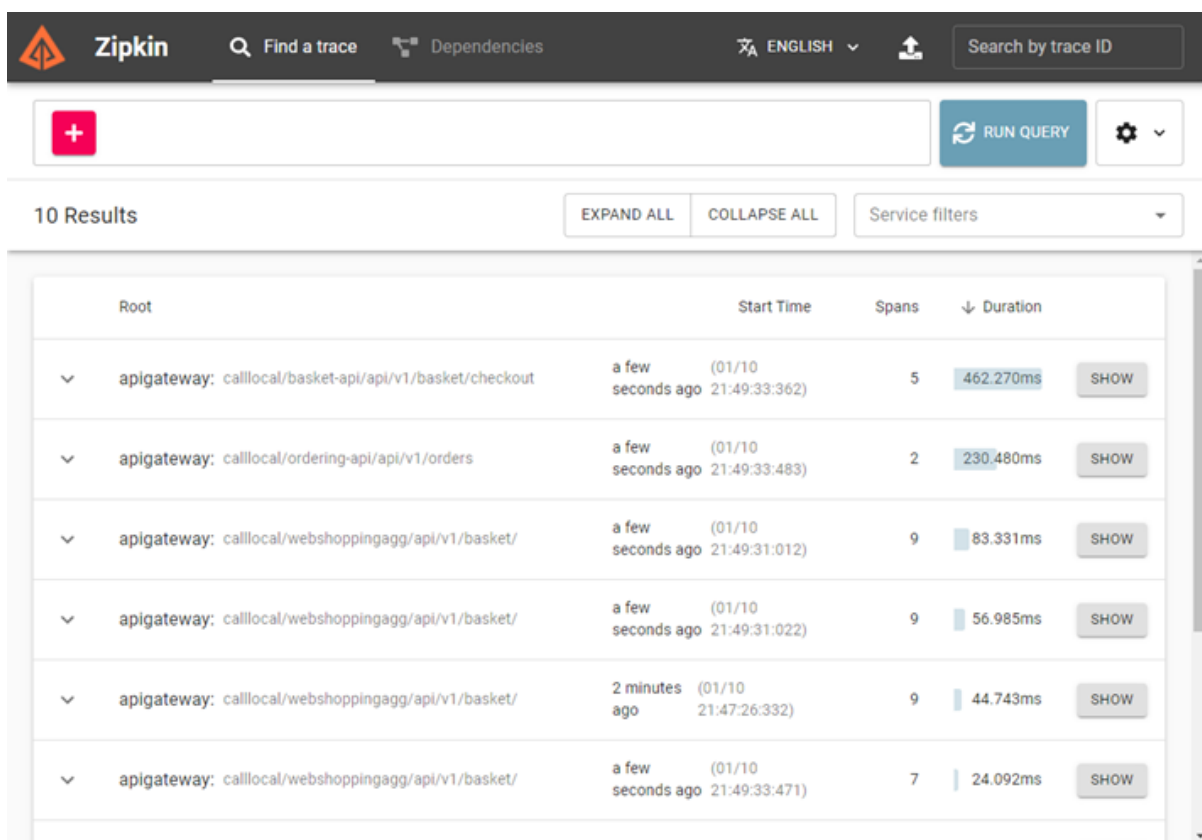
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ordering-api
  namespace: eshop
  labels:
    app: eshop
spec:
  replicas: 1
  selector:
    matchLabels:
      app: eshop
  template:
    metadata:
      labels:
        app: simulation
      annotations:
        dapr.io/enabled: "true"
        dapr.io/app-id: "ordering-api"
        dapr.io/config: "tracing-config"
    spec:
      containers:
        - name: simulation
          image: eshop/ordering.api:linux-latest
```

Inspect the telemetry in Zipkin

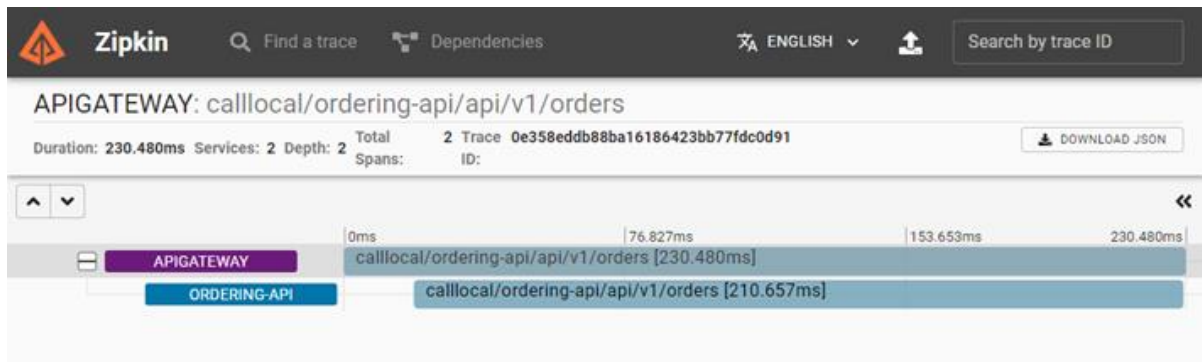
Once the application is started, the Dapr sidecars will emit telemetry to the Zipkin server. To inspect this telemetry, point a web-browser to <http://localhost:32411>. You'll see the Zipkin web front end:



On the *Find a trace* tab, you can query traces. Pressing the *RUN QUERY* button without specifying any restrictions will show all the ingested *traces*:



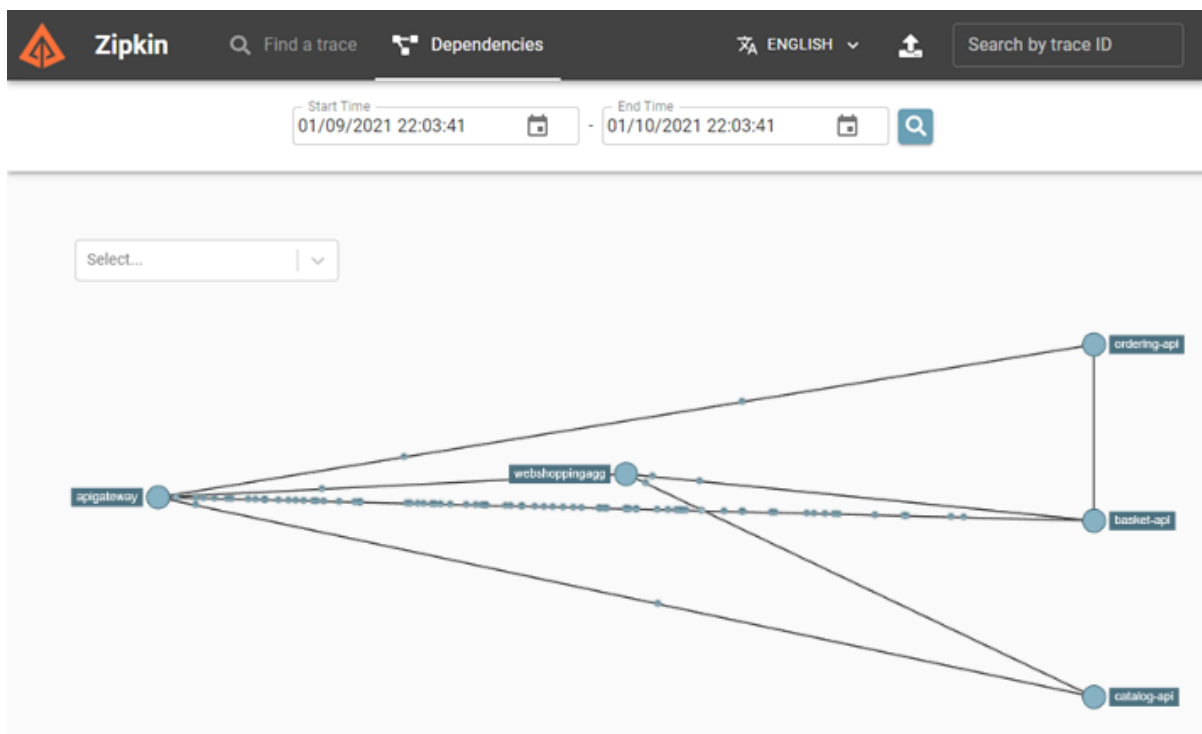
Clicking the *SHOW* button next to a specific trace, will show the details of that trace:



Each item on the details page, is a span that represents a request that is part of the selected trace.

Inspect the dependencies between services

Because Dapr sidecars handle traffic between services, Zipkin can use the trace information to determine the dependencies between the services. To see it in action, go to the *Dependencies* tab on the Zipkin web page and select the button with the magnifying glass. Zipkin will show an overview of the services and their dependencies:



The animated dots on the lines between the services represent requests and move from source to destination. Red dots indicate a failed request.

Use a Jaeger or New Relic monitoring back end

Beyond Zipkin itself, other monitoring back-end software also supports ingesting telemetry using the Zipkin format. [Jaeger](#) is an open source tracing system created by Uber Technologies. It's used to trace transactions between distributed services and troubleshoot complex microservices

environments. [New Relic](#) is a *full-stack* observability platform. It links relevant data from a distributed application to provide a complete picture of your system. To try them out, specify an `endpointAddress` pointing to either a Jaeger or New Relic server in the Dapr configuration file. Here's an example of a configuration file that configures Dapr to send telemetry to a Jaeger server. The URL for Jaeger is identical to the URL for the Zipkin. The only difference is the port on which the server runs:

```
apiVersion: dapr.io/v1alpha1
kind: Configuration
metadata:
  name: tracing-config
  namespace: default
spec:
  tracing:
    samplingRate: "1"
    zipkin:
      endpointAddress: "http://localhost:9415/api/v2/spans"
```

To try out New Relic, specify the endpoint of the New Relic API. Here's an example of a configuration file for New Relic:

```
apiVersion: dapr.io/v1alpha1
kind: Configuration
metadata:
  name: tracing-config
  namespace: default
spec:
  tracing:
    samplingRate: "1"
    zipkin:
      endpointAddress: "https://trace-api.newrelic.com/trace/v1?Api-Key=<NR-API-KEY>&Data-Format=zipkin&Data-Format-Version=2"
```

Check out the Jaeger and New Relic websites for more information on how to use them.

Metrics

Metrics provide insight into performance and resource consumption. Under the hood, Dapr emits a wide collection of system and runtime metrics. Dapr uses [Prometheus](#) as a metric standard. Dapr sidecars and system services, expose a metrics endpoint on port 9090. A *Prometheus scraper* calls this endpoint at a predefined interval to collect metrics. The scraper sends metric values to a monitoring back end. Figure 9-4 shows the scraping process:

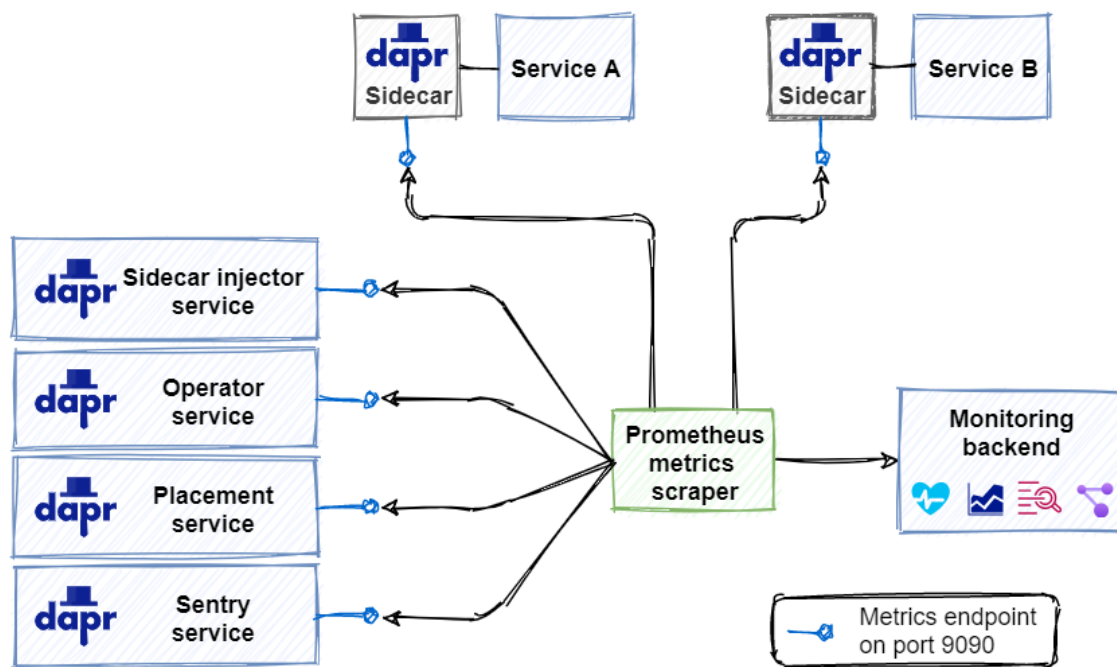


Figure 9-4. Scraping Prometheus metrics.

In the above figure, each sidecar and system service exposes a metric endpoint that listens on port 9090. The Prometheus Metrics Scraper captures metrics from each endpoint and published the information to the monitoring back end.

Service discovery

You might wonder how the metrics scraper knows where to collect metrics. Prometheus can integrate with discovery mechanisms built into target deployment environments. For example, when running in Kubernetes, Prometheus can integrate with the Kubernetes API to find all available Kubernetes resources running in the environment.

Metrics list

Dapr generates a large set of metrics for Dapr system services and its runtime. Some examples include:

Metric	Source	Description
dapr_operator_service_created_total	System	The total number of Dapr services created by the Dapr Operator service.
dapr_injector_sidecar_injection/requests_total	System	The total number of sidecar injection requests received by the Dapr Sidecar-Injector service.

Metric	Source	Description
dapr_placement_runtimes_total	System	The total number of hosts reported to the Dapr Placement service.
dapr_sentry_cert_sign_request_received_total	System	The number of certificate signing requests (CRSs) received by the Dapr Sentry service.
dapr_runtime_component_loaded	Runtime	The number of successfully loaded Dapr components.
dapr_grpc_io_server_completed_rpcs	Runtime	Count of gRPC calls by method and status.
dapr_http_server_request_count	Runtime	Number of HTTP requests started in an HTTP server.
dapr_http/client/sent_bytes	Runtime	Total bytes sent in request body (not including headers) by an HTTP client.

For more information on available metrics, see the [Dapr metrics documentation](#).

Configure Dapr metrics

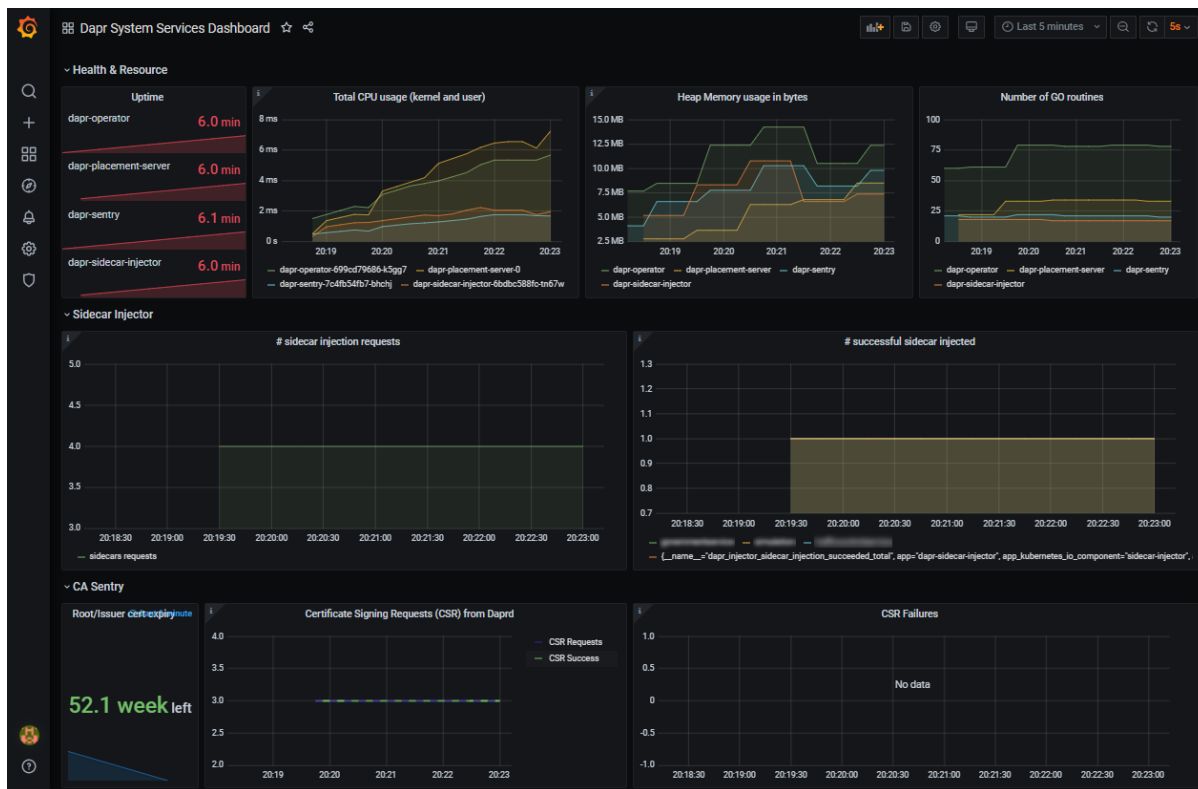
At runtime, you can disable the metrics collection endpoint by including the `--enable-metrics=false` argument in the Dapr command. Or, you can also change the default port for the endpoint with the `--metrics-port 9090` argument.

You can also use a Dapr configuration file to statically enable or disable runtime metrics collection:

```
apiVersion: dapr.io/v1alpha1
kind: Configuration
metadata:
  name: dapr-config
  namespace: eshop
spec:
  tracing:
    samplingRate: "1"
  metric:
    enabled: false
```

Visualize Dapr metrics

With the Prometheus scraper collecting and publishing metrics into the monitoring back end, how do you make sense of the raw data? A popular visualization tool for analyzing metrics is [Grafana](#). With Grafana, you can create dashboards from the available metrics. Here's an example of a dashboard displaying Dapr system services metrics:



The Dapr documentation includes a [tutorial for installing Prometheus and Grafana](#).

Logging

Logging provides insight into what is happening with a service at runtime. When running an application, Dapr automatically emits log entries from Dapr sidecars and Dapr system services. However, logging entries instrumented in your application code **aren't** automatically included. To emit logging from application code, you can import a specific SDK like [OpenTelemetry SDK for .NET](#). Logging application code is covered later in this chapter in the section *Using the Dapr .NET SDK*.

Log entry structure

Dapr emits structured logging. Each log entry has the following format:

Field	Description	Example
time	ISO8601 formatted timestamp	2021-01-10T14:19:31.000Z
level	Level of the entry (debug info warn error)	info
type	Log Type	log
msg	Log Message	metrics server started on :62408/
scope	Logging Scope	dapr.runtime
instance	Hostname where Dapr runs	TSTSRV01
app_id	Dapr App ID	ordering-api

Field	Description	Example
ver	Dapr Runtime Version	1.0.0-rc.2

When searching through logging entries in a troubleshooting scenario, the time and level fields are especially helpful. The time field orders log entries so that you can pinpoint specific time periods. When troubleshooting, log entries at the *debug* level provide more information on the behavior of the code.

Plain text versus JSON format

By default, Dapr emits structured logging in plain-text format. Every log entry is formatted as a string containing key/value pairs. Here's an example of logging in plain text:

```
== DAPR == time="2021-01-12T16:11:39.4669323+01:00" level=info msg="starting Dapr Runtime -
- version 1.0.0-rc.2 -- commit 196483d" app_id=ordering-api instance=TSTSRV03
scope=dapr.runtime type=log ver=1.0.0-rc.2
== DAPR == time="2021-01-12T16:11:39.467933+01:00" level=info msg="log level set to: info"
app_id=ordering-api instance=TSTSRV03 scope=dapr.runtime type=log ver=1.0.0-rc.2
== DAPR == time="2021-01-12T16:11:39.467933+01:00" level=info msg="metrics server started
on :62408/" app_id=ordering-api instance=TSTSRV03 scope=dapr.metrics type=log ver=1.0.0-
rc.2
```

While simple, this format is difficult to parse. If viewing log entries with a monitoring tool, you'll want to enable JSON formatted logging. With JSON entries, a monitoring tool can index and query individual fields. Here's the same log entries in JSON format:

```
{"app_id": "ordering-api", "instance": "TSTSRV03", "level": "info", "msg": "starting Dapr
Runtime -- version 1.0.0-rc.2 -- commit 196483d", "scope": "dapr.runtime", "time": "2021-
01-12T16:11:39.4669323+01:00", "type": "log", "ver": "1.0.0-rc.2"}
{"app_id": "ordering-api", "instance": "TSTSRV03", "level": "info", "msg": "log level set
to: info", "scope": "dapr.runtime", "type": "log", "time": "2021-01-
12T16:11:39.467933+01:00", "ver": "1.0.0-rc.2"}
{"app_id": "ordering-api", "instance": "TSTSRV03", "level": "info", "msg": "metrics server
started on :62408/", "scope": "dapr.metrics", "type": "log", "time": "2021-01-
12T16:11:39.467933+01:00", "ver": "1.0.0-rc.2"}
```

To enable JSON formatting, you need to configure each Dapr sidecar. In self-hosted mode, you can specify the flag `--log-as-json` on the command line:

```
dapr run --app-id ordering-api --log-level info --log-as-json dotnet run
```

In Kubernetes, you can add a `dapr.io/log-as-json` annotation to each deployment for the application:

```
annotations:
  dapr.io/enabled: "true"
  dapr.io/app-id: "ordering-api"
  dapr.io/app-port: "80"
  dapr.io/config: "dapr-config"
  dapr.io/log-as-json: "true"
```

When you install Dapr in a Kubernetes cluster using Helm, you can enable JSON formatted logging for all the Dapr system services:

```
helm repo add dapr https://dapr.github.io/helm-charts/
helm repo update
```

```
kubectl create namespace dapr-system
helm install dapr dapr/dapr --namespace dapr-system --set global.logAsJson=true
```

Collect logs

The logs emitted by Dapr can be fed into a monitoring back end for analysis. A log collector is a component that collects logs from a system and sends them to a monitoring back end. A popular log collector is [Fluentd](#). Check out the [How-To: Set up Fluentd, Elastic search and Kibana in Kubernetes](#) in the Dapr documentation. This article contains instructions for setting up Fluentd as log collector and the [ELK Stack](#) (Elastic Search and Kibana) as a monitoring back end.

Health status

The health status of a service provides insight into its availability. Each Dapr sidecar exposes a health API that can be used by the hosting environment to determine the health of the sidecar. The API has one operation:

```
GET http://localhost:3500/v1.0/healthz
```

The operation returns two HTTP status codes:

- 204: When the sidecar is healthy
- 500: when the sidecar isn't healthy

When running in self-hosted mode, the health API isn't automatically invoked. You can invoke the API though from application code or a health monitoring tool.

When running in Kubernetes, the Dapr sidecar-injector automatically configures Kubernetes to use the health API for executing *liveness probes* and *readiness probes*.

Kubernetes uses liveness probes to determine whether a container is up and running. If a liveness probe returns a failure code, Kubernetes will assume the container is dead and automatically restart it. This feature increases the overall availability of your application.

Kubernetes uses readiness probes to determine whether a container is ready to start accepting traffic. A pod is considered ready when all of its containers are ready. Readiness determines whether a Kubernetes service can direct traffic to a pod in a load-balancing scenario. Pods that aren't ready are automatically removed from the load-balancer.

Liveness and readiness probes have several configurable parameters. Both are configured in the container spec section of a pod's manifest file. By default, Dapr uses the following configuration for each sidecar container:

```
livenessProbe:
  httpGet:
    path: v1.0/healthz
    port: 3500
  initialDelaySeconds: 5
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 3
readinessProbe:
  httpGet:
```

```
path: v1.0/healthz
port: 3500
initialDelaySeconds: 5
periodSeconds: 10
timeoutSeconds : 5
failureThreshold: 3
```

The following parameters are available for the probes:

- The path specifies the Dapr health API endpoint.
- The port specifies the Dapr health API port.
- The initialDelaySeconds specifies the number of seconds Kubernetes will wait before it starts probing a container for the first time.
- The periodSeconds specifies the number of seconds Kubernetes will wait between each probe.
- The timeoutSeconds specifies the number of seconds Kubernetes will wait on a response from the API before timing out. A timeout is interpreted as a failure.
- The failureThreshold specifies the number of failed status code Kubernetes will accept before considering the container not alive or not ready.

Dapr dashboard

Dapr offers a dashboard that presents status information on Dapr applications, components, and configurations. Use the Dapr CLI to start the dashboard as a web-application on the local machine on port 8080:

```
dapr dashboard
```

For Dapr application running in Kubernetes, use the following command:

```
dapr dashboard -k
```

The dashboard opens with an overview of all services in your application that have a Dapr sidecar. The following screenshot shows the Dapr dashboard for the eShopOnDapr application running in Kubernetes:

dapr Dashboard Scope: eshop

Overview

Dapr Control Plane

Version: 1.0.0-rc.2
Status: Healthy ✓
[More Information](#)

Dapr Applications




Name	Labels	Status	Age	Selector
apigateway	app:eshop	1/1	1m	app:eshop
backgroundtasks	app:eshop	1/1	1m	app:eshop
basket-api	app:eshop	1/1	1m	app:eshop
catalog-api	app:eshop	1/1	1m	app:eshop
ordering-api	app:eshop	1/1	1m	app:eshop

The Dapr dashboard is invaluable when troubleshooting a Dapr application. It provides information about Dapr sidecars and system services. You can drill down into the configuration of each service, including the logging entries.

The dashboard also shows the configured components (and their configuration) for your application:

dapr Dashboard Scope: eshop

Dapr Components

Name	Type	Age	Created
 eshop-basket-statestore	state.redis	1m	2021-01-26 06:04:47
 pubsub	pubsub.redis	1m	2021-01-26 06:04:47
 sendmail	bindings.twilio.sendgrid	1m	2021-01-26 06:04:47

There's a large amount of information available through the dashboard. You can discover it by running a Dapr application and browsing the dashboard. You can use the accompanying eShopOnDapr application to start.

Check out the [Dapr dashboard CLI command reference](#) in the Dapr docs for more information on the Dapr dashboard commands.

Use the Dapr .NET SDK

The Dapr .NET SDK doesn't contain any specific observability features. All observability features are offered at the Dapr level.

If you want to emit telemetry from your .NET application code, you should consider the [OpenTelemetry SDK for .NET](#). The Open Telemetry project is cross-platform, open source, and vendor agnostic. It provides an end-to-end implementation to generate, emit, collect, process, and export telemetry data. There's a single instrumentation library per language that supports automatic and manual instrumentation. Telemetry is published using the Open Telemetry standard. The project has broad industry support and adoption from cloud providers, vendors, and end users.

Reference application: eShopOnDapr

Observability in accompanying eShopOnDapr reference application consists of several parts. Telemetry from all of the sidecars is captured. Additionally, there are other observability features inherited from the earlier eShopOnContainers sample.

Custom health dashboard

The **WebStatus** project in eShopOnDapr is a custom health dashboard that gives insight into the health of the eShop services. This dashboard doesn't use the Dapr health API but uses the built-in [health checks mechanism](#) of ASP.NET Core. The dashboard not only provides the health status of the services, but also the health of the dependencies of the services. For example, a service that uses a database also provides the health status of this database as shown in the following screenshot:

The screenshot shows the 'Health Checks status' dashboard. On the left is a sidebar with a menu icon, a profile picture, and two items: 'Health Checks' (selected) and 'Webhooks'. The main area has a title 'Health Checks status', a 'Refresh every 10 seconds' button, and a 'Change' button. Below this is a table of services with columns: NAME, HEALTH, ON STATE FROM, and LAST EXECUTION. The services listed are Basket API, Catalog API, Identity API, and Ordering API, all showing a 'Healthy' status. Below this is a detailed view of the 'Ordering' service, showing its components: OrderingDB-check, self, Ordering Background Tasks, Ordering SignalR Hub, Payment API, Web Shopping Aggregator, and Web SPA, all showing a 'Healthy' status.

NAME	HEALTH	ON STATE FROM	LAST EXECUTION
+ Basket API	✓ Healthy	Healthy a few seconds ago	1/26/2021, 10:38:00 PM
+ Catalog API	✓ Healthy	Healthy a few seconds ago	1/26/2021, 10:38:00 PM
+ Identity API	✓ Healthy	Healthy a few seconds ago	1/26/2021, 10:38:00 PM
- Ordering API	✓ Healthy	Healthy a few seconds ago	1/26/2021, 10:38:00 PM

NAME	HEALTH	DESCRIPTION	DURATION	DETAILS
OrderingDB-check	✓ Healthy		00:00:00.0087355	🔄
self	✓ Healthy		00:00:00.0000726	🔄
+ Ordering Background Tasks	✓ Healthy	Healthy a few seconds ago		1/26/2021, 10:38:00 PM
+ Ordering SignalR Hub	✓ Healthy	Healthy a minute ago		1/26/2021, 10:38:00 PM
+ Payment API	✓ Healthy	Healthy a minute ago		1/26/2021, 10:38:00 PM
+ Web Shopping Aggregator	✓ Healthy	Healthy a few seconds ago		1/26/2021, 10:38:00 PM
+ Web SPA	✓ Healthy	Healthy a few seconds ago		1/26/2021, 10:38:00 PM

Seq log aggregator

[Seq](#) is a popular log aggregator server that is used in eShopOnDapr to aggregate logs. Seq ingests logging from application services, but not from Dapr system services or sidecars. Seq indexes application logging and offers a web front end for analyzing and querying the logs. It also offers functionality for building monitoring dashboards.

The eShopOnDapr application services emit structured logging using the [Serilog](#) logging library. Serilog publishes log events to a construct called a **sink**. A sink is simply a target platform to which Serilog writes its logging events. [Many Serilog sinks are available](#), including one for Seq. Seq is the Serilog sink used in eShopOnDapr.

Application Insights

eShopOnDapr services also send telemetry directly to Azure Application Insights using the Microsoft Application Insights SDK for .NET Core. For more information, see [Azure Application Insights for ASP.NET Core applications](#) in the Microsoft docs.

Summary

Good observability is crucial when running a distributed system in production.

Dapr provides different types of telemetry, including distributed tracing, logging, metrics, and health status.

Dapr only produces telemetry for the Dapr system services and sidecars. Telemetry from your application code isn't automatically included. You can however use a specific SDK like the OpenTelemetry SDK for .NET to emit telemetry from your application code.

Dapr telemetry is produced in an open-standards based format so it can be ingested by a large set of available monitoring tools. Some examples are: Zipkin, Azure Application Insights, the ELK Stack, New Relic, and Grafana. See [Monitor your application with Dapr](#) in the Dapr documentation for tutorials on how to monitor your Dapr applications with specific monitoring back ends.

You'll need a telemetry scraper that ingests telemetry and publishes it to the monitoring back end.

Dapr can be configured to emit structured logging. Structured logging is favored as it can be indexed by back-end monitoring tools. Indexed logging enables users to execute rich queries when searching through the logging.

Dapr offers a dashboard that presents information about the Dapr services and configuration.

References

- [Azure Application Insights](#)
- [Open Telemetry](#)
- [Zipkin](#)
- [W3C Trace Context](#)
- [Jaeger](#)
- [New Relic](#)
- [Prometheus](#)
- [Grafana](#)
- [Open Telemetry SDK for .NET](#)
- [Fluentd](#)
- [ELK stack](#)
- [Seq](#)
- [Serilog](#)

The Dapr secrets building block

Enterprise applications require secrets. Common examples include:

- A database connection string that contains a username and password.
- An API key for calling an external web API.
- A client certificate for authenticating to an external system.

Secrets must be carefully managed so that they're never disclosed outside of the application.

Not long ago, it was popular to store application secrets in a configuration file inside the application codebase. .NET developers will fondly recall the *web.config* file. While simple to implement, integrating secrets to along with code was far from secure. A common misstep was to include the file when pushing to a public GIT repository, exposing the secrets to the world.

A widely accepted methodology for constructing modern distributed applications is [The Twelve-Factor App](#). It describes a set of principles and best practices. Its third factor prescribes that *configuration and secrets be externalized outside of the code base*.

To address this concern, the .NET Core platform includes a [Secret Manager](#) feature that stores sensitive data in a physical folder outside of the project tree. While secrets are outside of source control, this feature doesn't encrypt data. It's designed for **development purposes** only.

A more modern and secure practice is to isolate secrets in a secrets management tool like **Hashicorp Vault** or **Azure Key Vault**. These tools enable you to store secrets externally, vary credentials across environments, and reference them from application code. However, each tool has its complexities and learning curve.

Dapr offers a building block that simplifies managing secrets.

What it solves

The Dapr [secrets building block](#) abstracts away the complexity of working with secrets and secret management tools.

- It hides the underlying plumbing through a unified interface.
- It supports various *pluggable* secret store components, which can vary between development and production.

- Applications don't require direct dependencies on secret store libraries.
- Developers don't require detailed knowledge of each secret store.

Dapr handles all of the above concerns.

Access to the secrets is secured through authentication and authorization. Only an application with sufficient rights can access secrets. Applications running in Kubernetes can also use its built-in secrets management mechanism.

How it works

Applications use the secrets building block in two ways:

- Retrieve a secret directly from the application block.
- Reference a secret indirectly from a Dapr component configuration.

Retrieving secrets directly is covered first. Referencing a secret from a Dapr component configuration file is addressed in a later section.

The application interacts with a Dapr sidecar when using the secrets building block. The sidecar exposes the secrets API. The API can be called with either HTTP or gRPC. Use the following URL to call the HTTP API:

```
http://localhost:<dapr-port>/v1.0/secrets/<store-name>/<name>?<metadata>
```

The URL contains the following segments:

- <dapr-port> specifies the port number upon which the Dapr sidecar is listening.
- <store-name> specifies the name of the Dapr secret store.
- <name> specifies the name of the secret to retrieve.
- <metadata> provides additional information for the secret. This segment is optional and metadata properties differ per secret store. For more information on metadata properties, see the [secrets API reference]**INTERNAL-LINK:([Secrets API reference | Dapr Docs](#))**.

Note

The above URL represents the native Dapr API call available to any development platform that supports HTTP or gRPC. Popular platforms like .NET, Java, and Go have their own custom APIs.

The JSON response contains the key and value of the secret.

Figure 10-1 shows how Dapr handles a request for the secrets API:

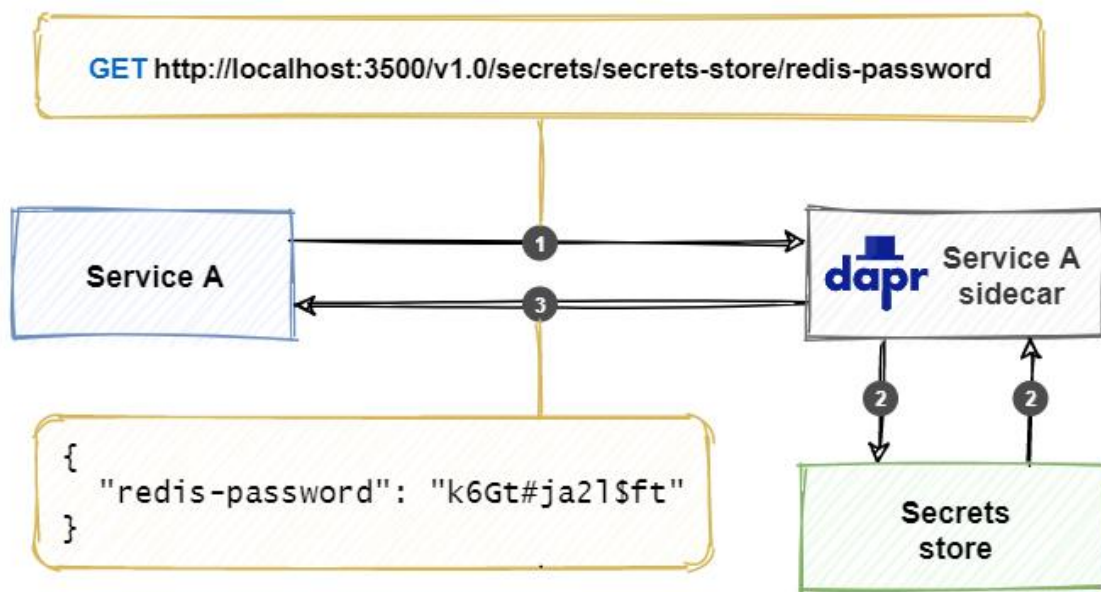


Figure 10-1. Retrieving a secret with the Dapr secrets API.

1. The service calls the Dapr secrets API, along with the name of the secret store, and secret to retrieve.
2. The Dapr sidecar retrieves the specified secret from the secret store.
3. The Dapr sidecar returns the secret information back to the service.

Some secret stores support storing multiple key/value pairs in a single secret. For those scenarios, the response would contain multiple key/value pairs in a single JSON response as in the following example:

```
GET http://localhost:3500/v1.0/secrets/secret-store/interestRates?metadata.version_id=3
```

```
{
  "tier1-percentage": "2.5",
  "tier2-percentage": "3.8",
  "tier3-percentage": "5.1"
}
```

The Dapr secrets API also offers an operation to retrieve all the secrets the application has access to:

```
http://localhost:<dapr-port>/v1.0/secrets/<store-name>/bulk
```

Use the Dapr .NET SDK

For .NET developers, the Dapr .NET SDK streamlines Dapr secret management. Consider the `DaprClient.GetSecretAsync` method. It enables you to retrieve a secret directly from any Dapr secret store with minimal effort. Here's an example of fetching a connection string secret for a SQL Server database:

```
var metadata = new Dictionary<string, string> { ["version_id"] = "3" };
Dictionary<string, string> secrets = await daprClient.GetSecretAsync("secret-store",
"eshopsecrets", metadata);
string connectionString = secrets["customerdb"];
```

Arguments for the GetSecretAsync method include:

- The name of the Dapr secret store component ('secret-store')
- The secret to retrieve ('eshopsecrets')
- Optional metadata key/value pairs ('version_id=3')

The method responds with a dictionary object as a secret can contain multiple key/value pairs. In the example above, the secret named customerdb is referenced from the collection to return a connection string.

The Dapr .NET SDK also features a .NET configuration provider. It loads specified secrets into the underlying [.NET Core configuration API](#). The running application can then reference secrets from the IConfiguration dictionary that is registered in ASP.NET Core dependency injection.

The secrets configuration provider is available from the [Dapr.Extensions.Configuration](#) NuGet package. The provider can be registered in the Program.cs of an ASP.NET Web API application:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration(config =>
        {
            var daprClient = new DaprClientBuilder().Build();
            var secretDescriptors = new List<DaprSecretDescriptor>
            {
                new DaprSecretDescriptor("eshopsecrets")
            };
            config.AddDaprSecretStore("secret-store", secretDescriptors, daprClient);
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

The above example loads the eshopsecrets secrets collection into the .NET configuration system at startup. Registering the provider requires an instance of DaprClient to invoke the secrets API on the Dapr sidecar. The other arguments include the name of the secret store and a DaprSecretDescriptor object with the name of the secret.

Once loaded, you can retrieve secrets directly from application code:

```
public void GetCustomer(IConfiguration config)
{
    var connectionString = config["eshopsecrets"]["customerdb"];
}
```

Secret store components

The secrets building block supports several secret store components. At the time of writing, the following secret stores are available:

- Environment Variables
- Local file
- Kubernetes secrets
- AWS Secrets Manager
- Azure Key Vault
- GCP Secret Manager
- HashiCorp Vault

Important

The environment variables and local file components are designed for development workloads only.

The following sections show how to configure a secret store.

Configuration

You configure a secret store using a Dapr component configuration file. The typical structure of the file is shown below:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: [component name]
  namespace: [namespace]
spec:
  type: secretstores.[secret store type]
  version: [secret store version]
  metadata:
    - name: [property name]
      value: [property value]
```

All Dapr component configuration files require a name along with an optional namespace value. Additionally, the type field in the spec section specifies the type of secret store component. The properties in the metadata section differ per secret store.

Indirectly consume Dapr secrets

As mentioned earlier in this chapter, applications can also consume secrets by referencing them in component configuration files. Consider a [state management component](#) that uses Redis cache for storing state:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: eshop-basket-statestore
  namespace: eshop
spec:
  type: state.redis
  version: v1
  metadata:
    - name: redisHost
      value: localhost:6379
```

```
- name: redisPassword
  value: e$h0p0nD@pr
```

The above configuration file contains a **clear-text** password for connecting to the Redis server. **Hardcoded** passwords are always a bad idea. Pushing this configuration file to a public repository would expose the password. Storing the password in a secret store would dramatically improve this scenario.

The following examples demonstrate this using several different secret stores.

Local file

The local file component is designed for development scenarios. It stores secrets on the local filesystem inside a JSON file. Here's an example named eshop-secrets.json. It contains a single secret - a password for Redis:

```
{
  "eShopRedisPassword": "e$h0p0nD@pr"
}
```

You place this file in a components folder that you specify when running the Dapr application.

The following secret store configuration file consumes the JSON file as a secret store. It's also placed in the components folder:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: eshop-local-secret-store
  namespace: eshop
spec:
  type: secretstores.local.file
  version: v1
  metadata:
    - name: secretsFile
      value: ./components/eshop-secrets.json
    - name: nestedSeparator
      value: ":"
```

The component type is secretstore.local.file. The secretsFile metadata element specifies the path to the secrets file.

Important

The path to a secrets file can be a absolute or relative path. The relative path is based on the folder in which the application starts. In the example, the components folder is a sub-folder of the directory that contains the .NET application.

From the application folder, start the Dapr application specifying the components path as a command-line argument:

```
dapr run --app-id basket-api --components-path ./components dotnet run
```

Note

This above example applies to running Dapr in self-hosted mode. For Kubernetes hosting, consider using volume mounts.

The `nestedSeparator` in a Dapr configuration file specifies a character to *flatten* a JSON hierarchy. Consider the following snippet:

```
{
  "redisPassword": "some password",
  "connectionStrings": {
    "customerdb": "some connection string",
    "productdb": "some connection string"
  }
}
```

Using a colon as a separator, you can retrieve the `customerdb` connection-string using the key `connectionStrings:customerdb`.

Note

The colon `:` is the default separator value.

In the next example, a state management configuration file references the local secret store component to obtain the password for connecting to the Redis server:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: eshop-basket-statestore
  namespace: eshop
spec:
  type: state.redis
  version: v1
  metadata:
    - name: redisHost
      value: localhost:6379
    - name: redisPassword
      secretKeyRef:
        name: eShopRedisPassword
        key: eShopRedisPassword
  auth:
    secretStore: eshop-local-secret-store
```

The `secretKeyRef` element references the secret containing the password. It replaces the earlier *clear-text* value. The secret name and the key name, `eShopRedisPassword`, reference the secret. The name of the secret management component `eshop-local-secret-store` is found in the `auth` metadata element.

You might wonder why `eShopRedisPassword` is identical for both the name and key in the secret reference. In the local file secret store, secrets aren't identified with a separate name. The scenario will be different in the next example using Kubernetes secrets.

Kubernetes secret

This second example focuses on a Dapr application running in Kubernetes. It uses the standard secrets mechanism that Kubernetes offers. Use the Kubernetes CLI (kubectl) to create a secret named eshop-redis-secret that contains the password:

```
kubectl create secret generic eshopsecrets --from-literal=redisPassword=e$h0p0nD@pr -n eshop
```

Once created, you can reference the secret in the component configuration file for state management:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: eshop-basket-statestore
  namespace: eshop
spec:
  type: state.redis
  version: v1
  metadata:
    - name: redisHost
      value: redis:6379
    - name: redisPassword
      secretKeyRef:
        name: eshopsecrets
        key: redisPassword
  auth:
    secretStore: kubernetes
```

The secretKeyRef element specifies the name of the Kubernetes secret and the secret's key, eshopsecrets, and redisPassword respectively. The auth metadata section instructs Dapr to use the Kubernetes secrets management component.

Note

Auth is the default value when using Kubernetes secrets and can be omitted.

In a production setting, secrets are typically created as part of an automated CI/CD pipeline. Doing so ensures only people with sufficient permissions can access and change the secrets. Developers create configuration files without knowing the actual value of the secrets.

Azure Key Vault

The next example is geared toward a real-world production scenario. It uses **Azure Key Vault** as the secret store. Azure Key Vault is a managed Azure service that enables secrets to be stored securely in the cloud.

For this example to work, the following prerequisites must be satisfied:

- You've secured administrative access to an Azure subscription.
- You've provisioned an Azure Key Vault named eshopkv that holds a secret named redisPassword that contains the password for connecting to the Redis server.

- You've created [service principal](#) in Azure Active Directory.
- You've installed an X509 certificate for this service principal (containing both the public and private key) on the local filesystem.

Note

A service principal is an identity that can be used by an application to authenticate an Azure service. The service principal uses a X509 certificate. The application uses this certificate as a credential to authenticate itself.

The [Dapr Azure Key Vault secret store documentation](#) provides step-by-step instructions to create and configure a Key Vault environment.

Use Key Vault when running in self-hosted mode

Consuming Azure Key Vault in Dapr self-hosted mode requires the following configuration file:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: eshop-azurekv-secret-store
  namespace: eshop
spec:
  type: secretstores.azure.keyvault
  version: v1
  metadata:
    - name: vaultName
      value: eshopkv
    - name: spnTenantId
      value: "619926af-a7c3-4e95-93ed-4ecc4e3e652b"
    - name: spnClientId
      value: "6cf48032-6c38-43be-9d6f-2a43ce736b09"
    - name: spnCertificateFile
      value: "azurekv-spn-cert.pfx"
```

The secret store type is `secretstores.azure.keyvault`. The metadata element to configure access to Key Vault requires the following properties:

- The `vaultName` contains the name of the Azure Key Vault.
- The `spnTenantId` contains the *tenant ID* of the service principal used to authenticate against the Key Vault.
- The `spnClientId` contains the *app ID* of the service principal used to authenticate against the Key Vault.
- The `spnCertificateFile` contains the path to the certificate file for the service principal to authenticate against the Key Vault.

Tip

You can copy the service principal information from the Azure portal or Azure CLI .

Now the application can retrieve the Redis password from the Azure Key Vault.

Use Key Vault when running on Kubernetes

Consuming Azure Key Vault with Dapr and Kubernetes also requires a service principal to authenticate against the Azure Key Vault.

First, create a *Kubernetes secret* that contains a certificate file using the `kubectl` CLI tool:

```
kubectl create secret generic [k8s_spn_secret_name] --from-  
file=[pfx_certificate_file_local_path] -n eshop
```

The command requires two command-line arguments:

- `[k8s_spn_secret_name]` is the secret name in Kubernetes secret store.
- `[pfx_certificate_file_local_path]` is the path of X509 certificate file.

Once created, you can reference the Kubernetes secret in the secret store component configuration file:

```
apiVersion: daprio.io/v1alpha1  
kind: Component  
metadata:  
  name: eshop-azurekv-secret-store  
  namespace: eshop  
spec:  
  type: secretstores.azure.keyvault  
  version: v1  
  metadata:  
    - name: vaultName  
      value: [your_keyvault_name]  
    - name: spnTenantId  
      value: "619926af-a7c3-4e95-93ed-4ecc4e3e652b"  
    - name: spnClientId  
      value: "6cf48032-6c38-43be-9d6f-2a43ce736b09"  
    - name: spnCertificate  
      secretKeyRef:  
        name: [k8s_spn_secret_name]  
        key: [pfx_certificate_file_local_name]  
  auth:  
    secretStore: kubernetes
```

At this point, an application running in Kubernetes can retrieve the Redis password from the Azure Key Vault.

Important

It's critical to keep the X509 certificate file for the service principal in a safe place. It's best to place it in a well-known folder outside the source-code repository. The configuration file can then reference the certificate file from this well-known folder. On a local development machine, you're responsible for copying the certificate to the folder. For automated deployments, the pipeline will copy the certificate to the machine where the application is deployed. It's a best practice to use a different service principal per environment. Doing so prevents the service principal from a DEVELOPMENT environment to access secrets in a PRODUCTION environment.

When running in Azure Kubernetes Service (AKS), it's preferable to use an [Azure Managed Identity](#) for authenticating against Azure Key Vault. Managed identities are outside of the scope of this book, but explained in the [Azure Key Vault with managed identities](#) documentation.

Scope secrets

Secret scopes allow you to control which secrets your application can access. You configure scopes in a Dapr sidecar configuration file. The [Dapr configuration documentation](#) provides instructions for scoping secrets.

Here's an example of a Dapr sidecar configuration file that contains secret scopes:

```
apiVersion: daprio.io/v1alpha1
kind: Configuration
metadata:
  name: dapr-config
  namespace: eshop
spec:
  tracing:
    samplingRate: "1"
  secrets:
    scopes:
      - storeName: eshop-azurekv-secret-store
        defaultAccess: allow
        deniedSecrets: ["redisPassword", "apiKey"]
```

You specify scopes per secret store. In the above example, the secret store is named eshop-azurekv-secret-store. You configure access to secrets using the following properties:

Property	Value	Description
defaultAccess	allow or deny	Allows or denies access to <i>all</i> secrets in the specified secret store. This property is optional with a default value of allow.
allowedSecrets	List of secret keys	Secrets specified in the array will be accessible. This property is optional.
deniedSecrets	List of secret keys	Secrets specified in the array will NOT be accessible. This property is optional.

The allowedSecrets and deniedSecrets properties take precedence over the defaultAccess property. Imagine specifying defaultAccess: allowed and an allowedSecrets list. In this case, only the secrets in the allowedSecrets list would be accessible by the application.

Reference application: eShopOnDapr

The eShopOnDapr reference application uses the secrets building block for two secrets:

- The password for connecting to the Redis cache.
- The API-key for using the Twilio Sendgrid API. The application uses Twilio to send emails using a Dapr output binding (as described in the [bindings building block chapter](#)).

When running the application using Docker Compose, the **local file** secret store is used. The component configuration file `eshop-secretstore.yaml` is found in the `dapr/components` folder of the `eShopOnDapr` repository:

```
apiVersion: daprio/v1alpha1
kind: Component
metadata:
  name: eshop-secretstore
  namespace: eshop
spec:
  type: secretstores.local.file
  version: v1
  metadata:
    - name: secretsFile
      value: ./components/eshop-secretstore.json
```

The configuration file references the local store file `eshop-secretstore.json` located in the same folder:

```
{
  "redisPassword": "*****",
  "sendgridAPIKey": "*****"
}
```

The components folder is specified in the command-line and mounted as a local folder inside the Dapr sidecar container. Here's a snippet from the `docker-compose.override.yml` file in the repository root that specifies the volume mount:

```
ordering-backgroundtasks-dapr:
  command: ["/dapr",
    "-app-id", "ordering-backgroundtasks",
    "-app-port", "80",
    "-dapr-grpc-port", "50004",
    "-components-path", "/components",
    "-config", "/configuration/eshop-config.yaml"
  ]
  volumes:
    - "/dapr/components:/components"
    - "/dapr/configuration:/configuration"
```

Note

The Docker Compose override file contains environmental specific configuration values.

The `/components` volume mount and `--components-path` command-line argument are passed into the `dapr` startup command.

Once configured, other component configuration files can also reference the secrets. Here's an example of the Publish/Subscribe component configuration consuming secrets:

```
apiVersion: daprio/v1alpha1
kind: Component
metadata:
  name: pubsub
  namespace: eshop
spec:
  type: pubsub.redis
```

```
version: v1
metadata:
- name: redisHost
  value: redis:6379
- name: redisPassword
  secretKeyRef:
    name: redisPassword
auth:
  secretStore: eshop-secretstore
```

In the preceding example, the local Redis store is used to reference secrets.

Summary

The Dapr secrets building block provides capabilities for storing and retrieving sensitive configuration settings like passwords and connection-strings. It keeps secrets private and prevents them from being accidentally disclosed.

The building block supports several different secret stores and hides their complexity with the Dapr secrets API.

The Dapr .NET SDK provides a `DaprClient` object to retrieve secrets. It also includes a .NET configuration provider that adds secrets to the .NET Core configuration system. Once loaded, you can consume these secrets in your .NET code.

You can use secret scopes to control access to specific secrets.

References

- [Beyond the Twelve-Factor Application](#)

Summary and the road ahead

We're at the end of our Dapr flight. The jet plane flying at 20,000 feet from [chapter 2](#) is on final approach and about to land.

As the plane taxis to the gate, let's take a minute to review some important conclusions from this guide:

- **Dapr** - Dapr is a *Distributed Application Runtime* that streamlines how you build distributed applications. It exposes an architecture of building blocks and pluggable components. Dapr provides a **dynamic glue** that binds your application with infrastructure capabilities that exist in the Dapr runtime. Instead of building infrastructure plumbing, you and your team focus on delivering business features to customers.
- **Open source and cross-platform** - The native Dapr API can be consumed by *any platform* that supports HTTP or gRPC. Dapr also provides language-specific SDKs for popular development platforms. Dapr v1.0 supports Go, Python, .NET, Java, PHP, and JavaScript.
- **Building blocks** - Dapr building blocks encapsulate distributed application functionality. At the time of this writing, Dapr supports the seven building blocks shown in figure 11-1.

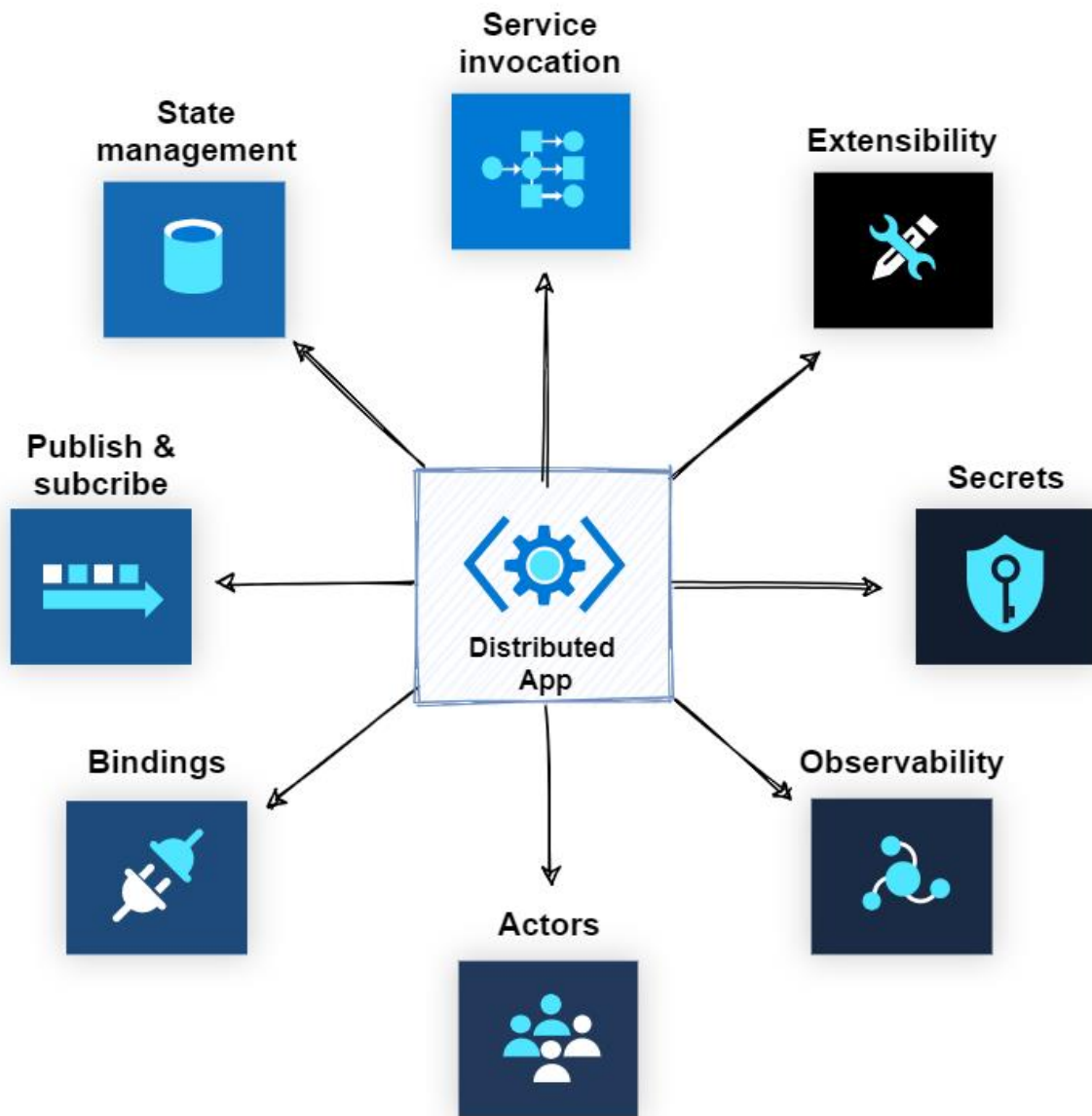


Figure 11-1. Dapr building blocks.

- **Components** - Dapr components provide the concrete implementation for each Dapr building block capability. They expose a common interface that enables developers to swap out component implementations without changing application code. Figure 11-2 shows the relationship among components, building blocks, and your service.

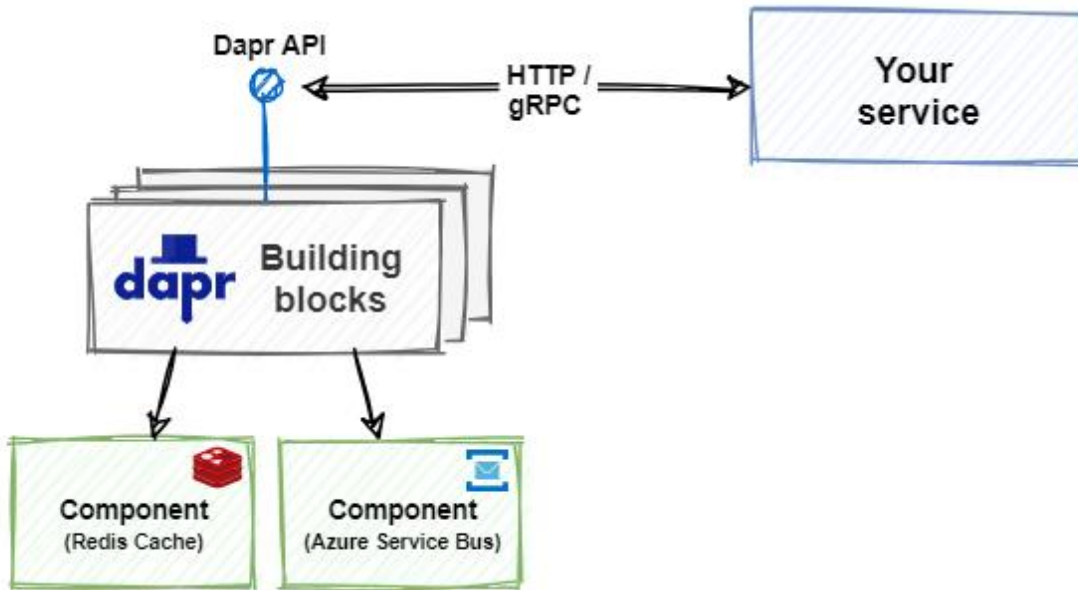


Figure 11-2. Dapr building block integration.

- **Sidecars** - Dapr runs alongside your application in a sidecar architecture, either as a separate process of a container. Your application communicates with the Dapr APIs over HTTP and gRPC. Sidecars provide isolation and encapsulation as they aren't part of the service, but connected to it. Figure 11-3 shows a sidecar architecture.

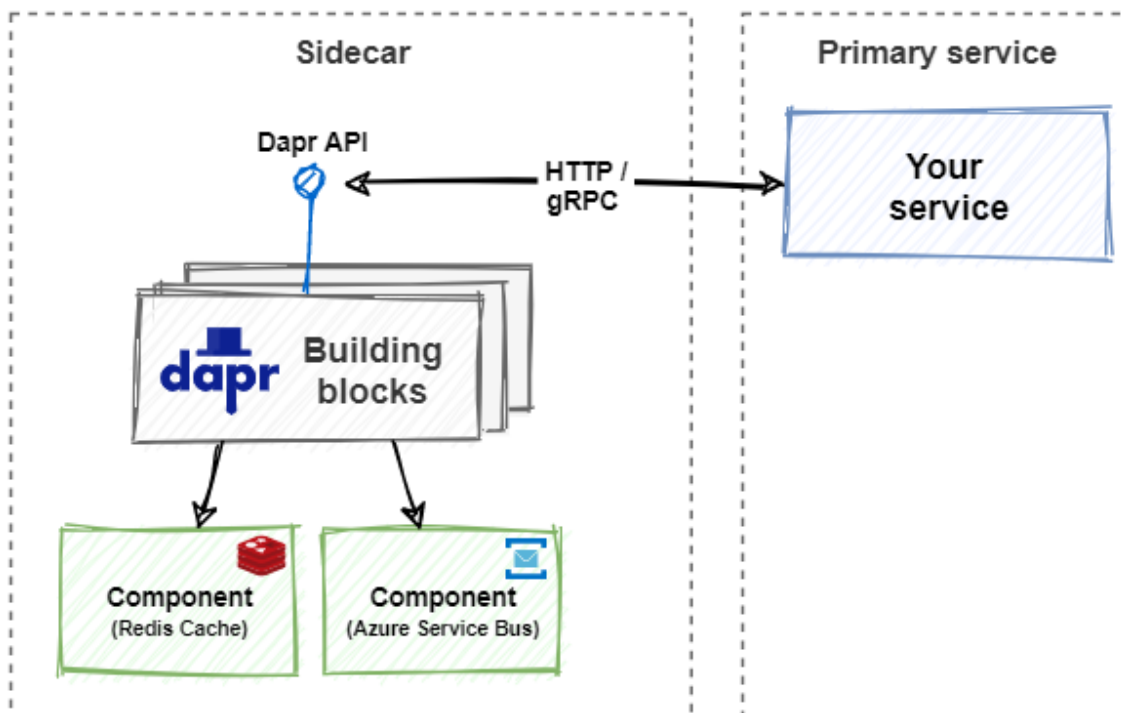


Figure 11-3. Sidecar architecture.

- **Hosting environments** Dapr has cross-platform support and can run in multiple environments. At the time of this writing, the environments include a local self-hosted mode and Kubernetes.
- **eShopOnDapr** - This book includes an accompanying reference application entitled [eShopOnDapr](#). Using a popular e-commerce application domain, the reference application demonstrates the usage of each building block. It's an evolution of the widely popular [eShopOnContainers](#), released several years ago.

The road ahead

Looking forward, Dapr has the potential to have a profound impact on distributed application development. What can you expect from the Dapr team and its open-source contributors?

At the time of writing, the list of proposed enhancements for Dapr include:

- Feature enhancements to existing building blocks:
 - Query capabilities in state management enabling you to retrieve multiple values.
 - Topic filtering in pub/sub enabling you to filter topics based on their content.
 - An application tracing API in observability that provides tracing in the application directly without having to bind to specific libraries.
 - Binding and pub/sub support for actors providing event driven capabilities to the actor programming model. Bound components will trigger events and messages invoke methods in the actor.
- New building blocks:
 - Configuration API building block for reading and writing configuration data. The block will bind to providers that include Azure Configuration Manager or GCP Configuration Management.
 - Http scale-to-zero autoscale.
 - Leader election building block to provide singleton instances and locking semantic capabilities.
 - Transparent proxying building block for service invocation, enabling you to route messages based on URLs or DNS addresses at the network level.
 - Resiliency building block (circuit breakers, bulkheads & timeouts).
- Integration with frameworks and cloud native technologies. Some examples include:
 - Django
 - Nodejs
 - Express
 - Kyma
 - Midway
- New language SDKs:
 - JavaScript
 - RUST

- C++
- New hosting platforms:
 - VMs
 - Azure IoT Edge
 - Azure Stack Edge
 - Azure Service Fabric
- Developer and operator productivity tooling:
 - VS Code extension.
 - Remote Dev Containers for local debugging a DevOps pipeline development.
 - Dapr operational dashboard enhancements that will provide deeper visibility into the operational concerns of managing Dapr applications.

Dapr version 1.0 provides developers with a compelling toolbox for building distributed applications. As the proposed enhancement list shows, Dapr is under active development with many new capabilities to come. Stay tuned to the [Dapr site](#) and [Dapr announcement blog](#) for future updates.