Microsoft

# Dapr for .NET Developers


Robert Vettor
Sander Molenkamp
Edwin van Wijk

Preview Edition

**EDITION v.1.0**

PUBLISHED BY

Microsoft Developer Division, .NET, and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399



Authors:

**Rob Vettor**, Principal Cloud System Architect/IP Architect - thinkingincloudnative.com, Microsoft

**Sander Molenkamp**, Principal Cloud Architect/Microsoft MVP - Info Support [])

**Edwin van Wijk**, Principal Software Architect/Microsoft MVP - Info Support [])

Participants and Reviewers:

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

**Mark Fussell**, Prinicpal Program Manager, Azure Incubations, Microsoft

**Yaron Schneider**, Prinicpal Software Engineer, Azure Incubations, Microsoft

**Ori Zohar**, Senior Program Manager, Azure Incubations, Microsoft

Editors:

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

# Version

This guide has been written to cover **.NET Core 3.1** version along with many additional updates related to the same "wave" of technologies (that is, Azure and additional third-party technologies) coinciding in time with the .NET Core 3.1 release.

# 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

# 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.

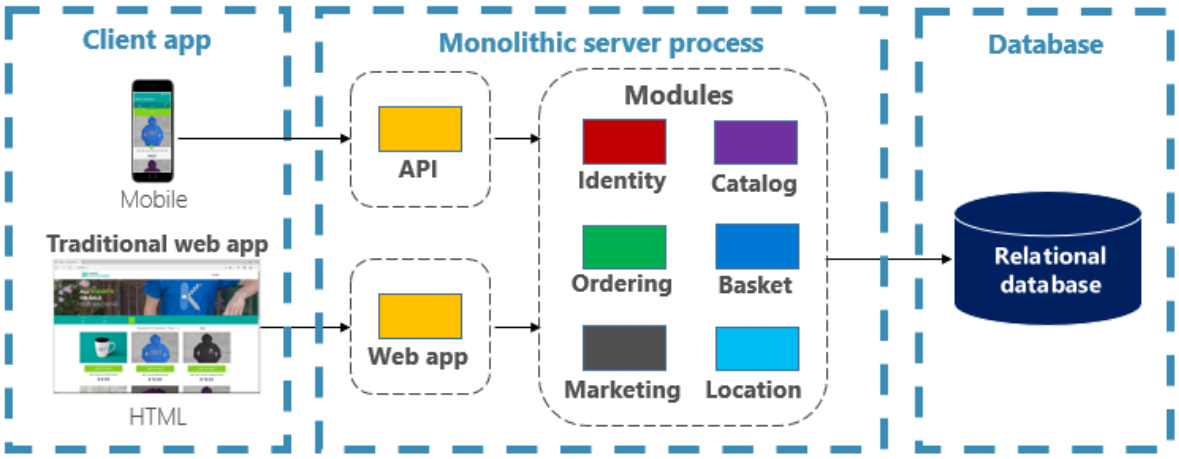


*Figure 1-1 : Monolithic architecture.*

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, the majority 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.

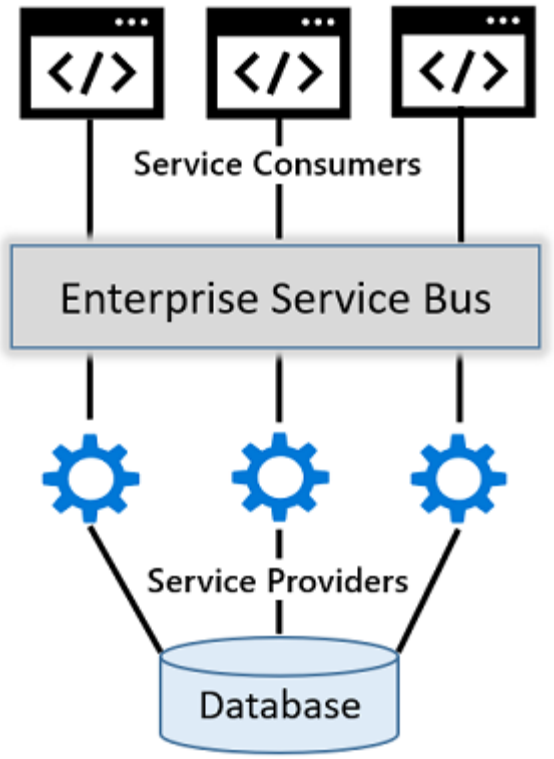


*Figure 1-2 SOA architecture.*

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.

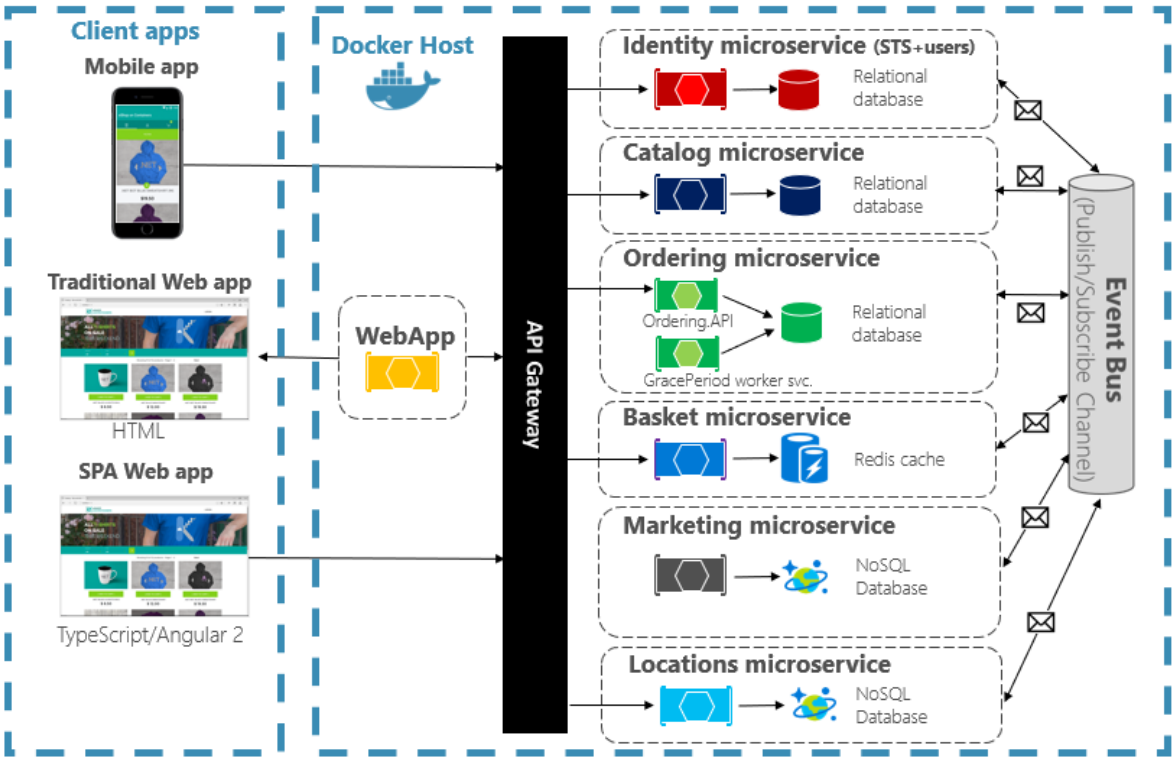


*Figure 1-3 : Distributed architecture.*

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 cannot 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 back-end 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.

## References

# 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 we "have our cake" and "eat it too?". That is, how can we take advantage of the agility of distributed architecture, and minimize the complexity?

Dapr, or *Distributed Application Runtime*, is a new way to build modern distributed applications while streamlining the underlying plumbing.

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, we help you build a solid conceptual understanding of Dapr and how it works. Later on, we present practical, hands-on instruction on how to use Dapr in your applications.

Imagine flying in a jet at 20,000 feet. You look out the window and see the landscape 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 helps simplify plumbing concerns. It enables your services to bind to distributed application capabilities. The runtime provides a **dynamic glue** that fuses your application with these capabilities. It does so *without* adding tightly coupled dependencies on infrastructure such as databases and message brokers. 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 greatly simplifies your experience by providing you with an abstraction out-of-the-box. You instruct your service to invoke a Dapr **building block** that dynamically binds to Redis Cache via a 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.

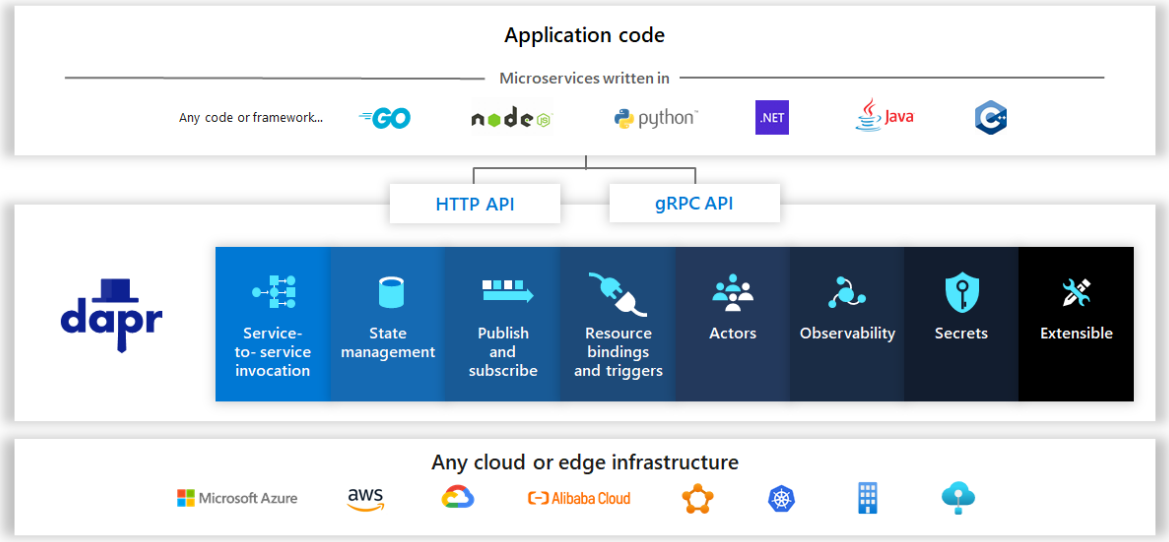


*Figure 2-1. Dapr at 20,000 feet.*

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 abstracts 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, our jet turns around and flies back over Dapr, descending in altitude, giving us a closer look at how Dapr works.

## Building blocks

From our new perspective, we have a more detailed view of the Dapr **building blocks**.

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

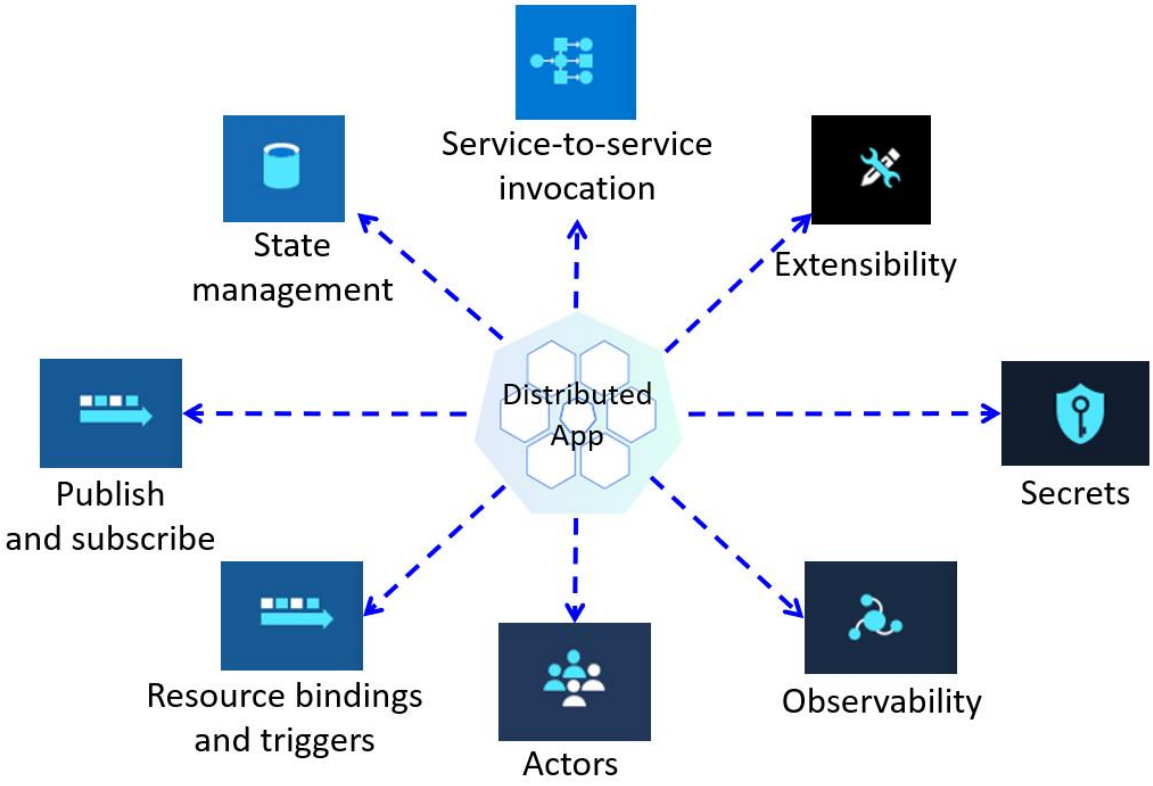


*Figure 2-2. Dapr building blocks.*

The following table describes the services provided by each block.

| Building Block | Description |
|---|---|
| Service-to-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. |
| State management | Support contextual information for long running stateful services. |
| Observability | Monitor and measure message calls across networked services. |
| Secrets | Securely access external secret stores. |
| Actors | Encapsulate logic and data in reusable actor objects. |
| Resource bindings and triggers | Trigger code from events raised by external resources with bi-directional communication. |

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

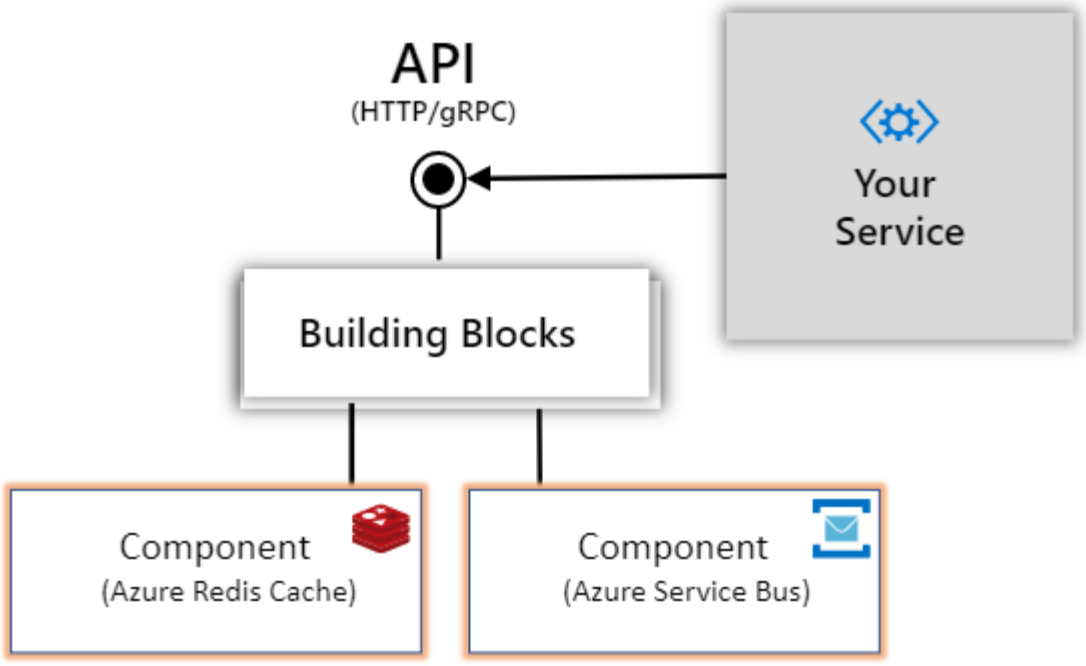


*Figure 2-3. Dapr building block integration.*

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.

We provide detailed explanation and code samples for each Dapr building block in the upcoming chapters. At this point, our jet descends even more. From our new perspective, we can 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:

:::{custom-style=CodeBox} go  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 interface above along with all of Dapr has been written in the Golang, or Go, platform. 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:

:::{custom-style=CodeBox} yaml  apiVersion: dapr.io/v1alpha1  kind: Component  metadata:  name: statestore  namespace: default  spec:  type: state.redis  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 our jet completes it fly over of Dapr, we 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.

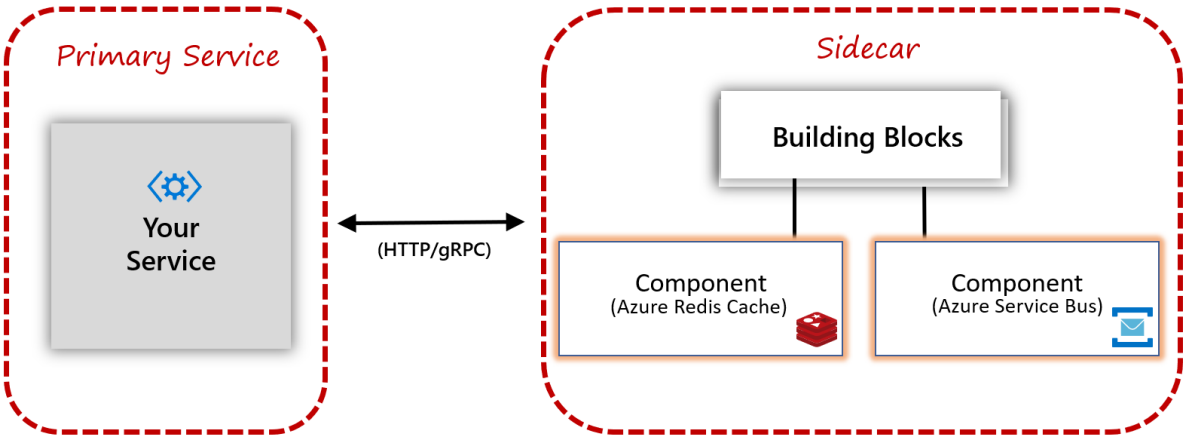


*Figure 2-4. Sidecar architecture.*

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.

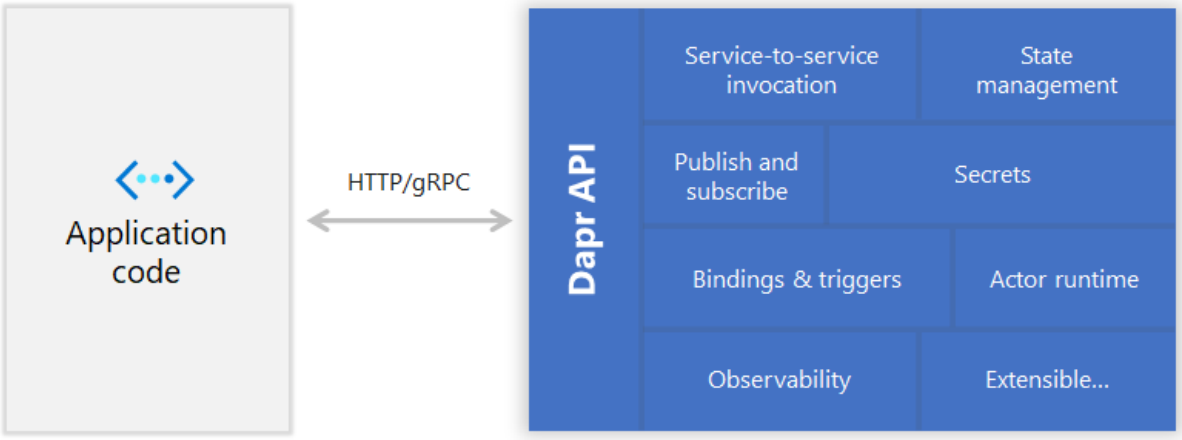


*Figure 2-5. Self-hosted Dapr sidecar*

By default, Dapr will install Docker containers for Redis and Zipkin to ensure building blocks such as state management and observability work out of the box. 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.

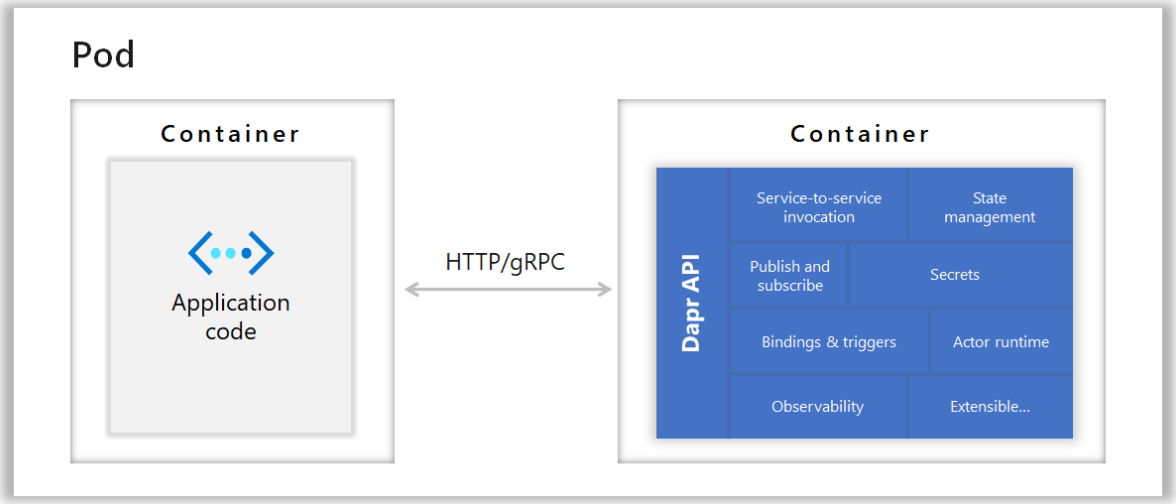


*Figure 2-6. Kubernetes-hosted Dapr sidecar*

# 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.

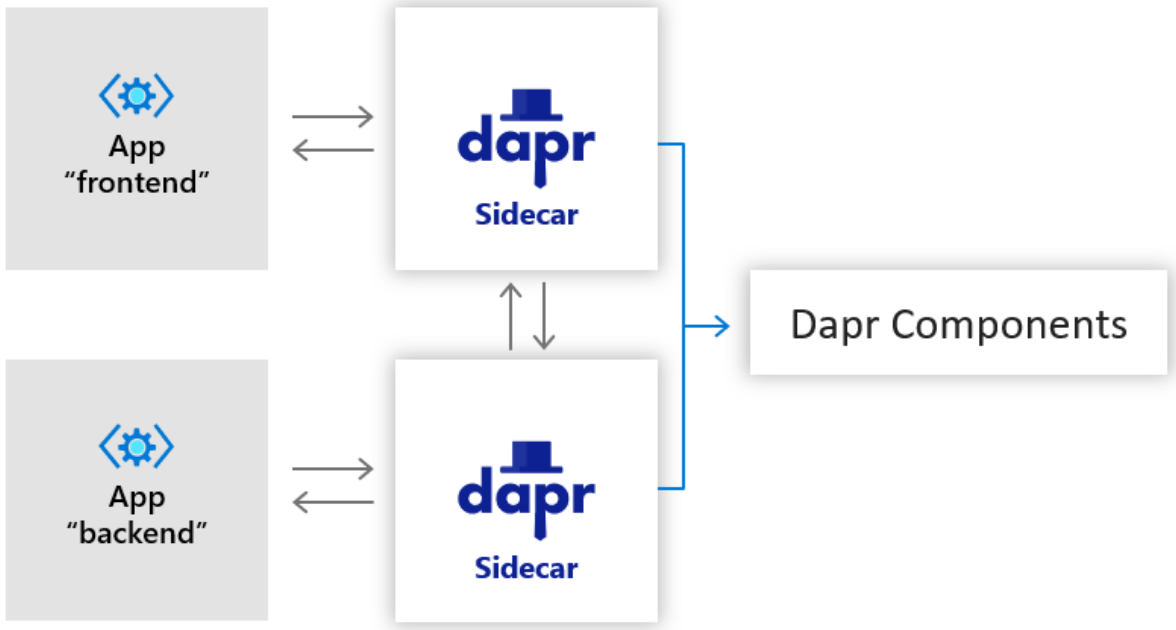


*Figure 2-7. Dapr traffic patterns*

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.

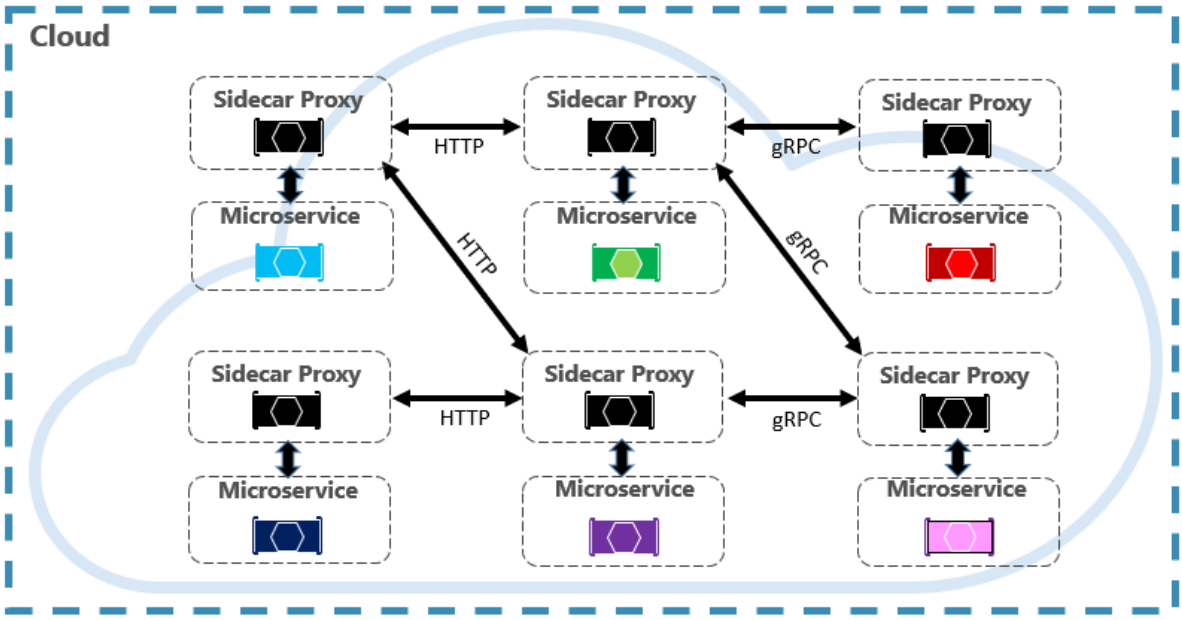


*Figure 2-8. Service mesh with a side car.*

The previous figure depicts 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.

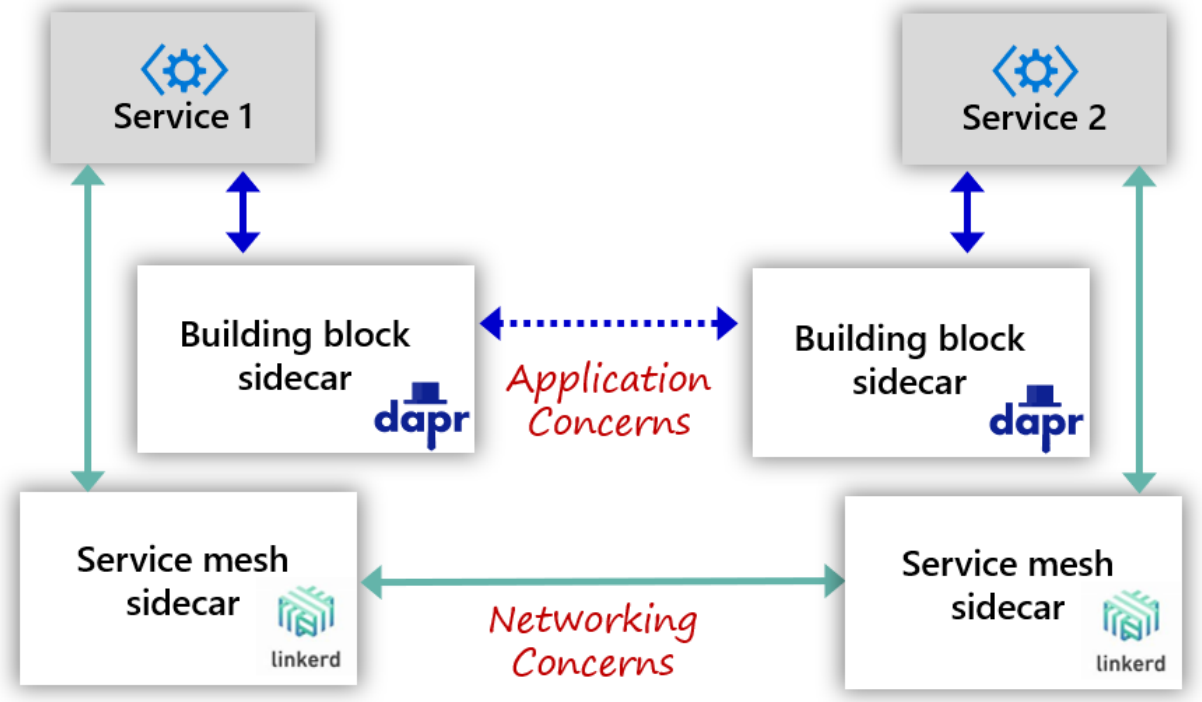


*Figure 2-9. Dapr and service mesh together.*

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