**Chapter 2. Communication Patterns**

Cloud native applications comprise a collection of microservices that are connected with one another, as well as external systems, through interservice communication techniques. With the proliferation of microservices and ever-increasing business requirements, building robust communication among microservices in cloud native applications is one of the hardest challenges in cloud native architectures.

This chapter will give you a broad understanding of the communication patterns and implementation technologies that you can use to build cloud native applications. These patterns can be used to build communication among microservices, other external systems, and consumer applications such as mobile and web apps. In this chapter, we focus mainly on foundational communication patterns for synchronous and asynchronous communication.

In *synchronous communication*, one microservice invokes another microservice and expects a response within a given time frame. For this, we use patterns such as Request-Response and Remote Procedure Calls (RPC). In *asynchronous communication*, microservices communicate by passing messages asynchronously with the help of an intermediary (known as a *message broker*), and we use patterns utilizing queue-based messaging and publisher-subscriber messaging. In most real-world cloud native applications, we can mix and match these communication patterns.

In addition to these foundational communication patterns, we’ll also explore patterns related to defining the service interfaces of your cloud native applications’ microservices so that the consumers of those microservices know how to interact with your service, either through synchronous or asynchronous communication techniques. Let’s begin our discussion with synchronous messaging patterns.

**Synchronous Messaging Patterns**

When building cloud native applications, you may discover that one microservice needs to invoke one or more other microservices, and then wait for a timely response to complete its business logic. To build such microservices, we can use *synchronous messaging patterns*, in which the business logic of a given microservice is dependent on one or more other microservices or systems. For instance, if you are building an online retail application, the Search microservice that you build has to accept queries and send the responses rapidly by invoking the relevant downstream microservices or systems.

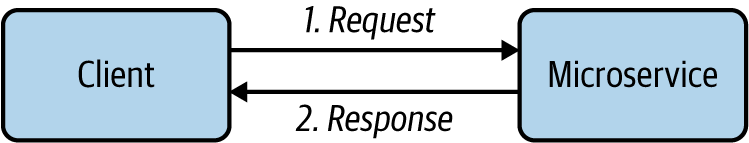
You can use multiple messaging patterns to build synchronous communication into your cloud native application. Let’s dive into one of the most commonly used patterns: Request-Response.

**Request-Response Pattern**

The *Request-Response pattern* is probably the most commonly used communication pattern in cloud native applications, as well as in the space of distributed computing at large. This pattern requires both parties to act in a timely fashion to pass data between them.

**How it works**

In the Request-Response pattern ([Figure 2-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#request_response_pattern-id00264)), one microservice (which acts as the client) sends a request and waits for a response from one or more other microservices or systems. The business logic of the client application blocks until it receives the response, and the communication channel has to be kept open until the response is received by the client application.



**Figure 2-1. Request-Response pattern**

The Request-Response pattern seeks to establish a connection between the client and the server application (microservice) and to exchange data between them in a synchronous way. The data exchanged is known as *messages.* Once a connection is established, the client sends a request for data to the microservice and waits until either it receives the response data, or the maximum time that the client intends to wait is reached (which is known as a *time-out*). Since this pattern is similar to querying a given entity, this style is sometimes referred to as a *query-based* interaction.

**How it’s used in practice**

The Request-Response pattern of communication is often used for building cloud native applications containing business logic that is interactive (frequent exchange of messages) and requires an immediate response. Most external-facing services are implemented using this pattern. Since it is agnostic of the underlying network protocols and the data format of the request and response message, we can use a wide range of techniques to realize this pattern.

Microservices designed using HTTP and RESTful services are the most common realization of this pattern, which was popular in the early stages of cloud native architecture. However, now quite a few request-response communication techniques are in use based on disparate use cases and requirements. We discuss them in the latter part of the chapter when we dive into the implementation technologies of cloud native communication patterns.

**Considerations**

This synchronous Request-Response pattern is the most commonly used pattern when building communication among microservices in cloud native applications. However, having a greater number of synchronous request-response interactions results in more coupling among microservices. A service that sends a request and expects a response creates an implicit dependency on the services that it calls. This approach works well for a small number of services, but as the number of services connected via request-response messaging increases, we create a chain of dependent services. Each service in the chain can introduce a potential performance bottleneck or downtime, which impacts all other services in the chain.

Therefore, you should select this pattern whenever you need interactive communication among microservices or with external consumers or systems. For other use cases, choose asynchronous communication (which we’ll explore later in this chapter). Often you will have to mix and match the Request-Response pattern with other communication patterns in real-world use cases.

**Related patterns**

The Request-Response communication pattern is commonly used with the following patterns:

*Service Orchestration and API Gateway*

These patterns heavily use the request-response style of communication when building the composition of services and exposing services as managed APIs, respectively.

*Request-Reply*

This pattern uses an intermediary to implement a similar style of communication with the use of a queue.

Now that you have a general understanding of Request-Response, let’s look at the Remote Procedure Calls pattern.

**Remote Procedure Calls Pattern**

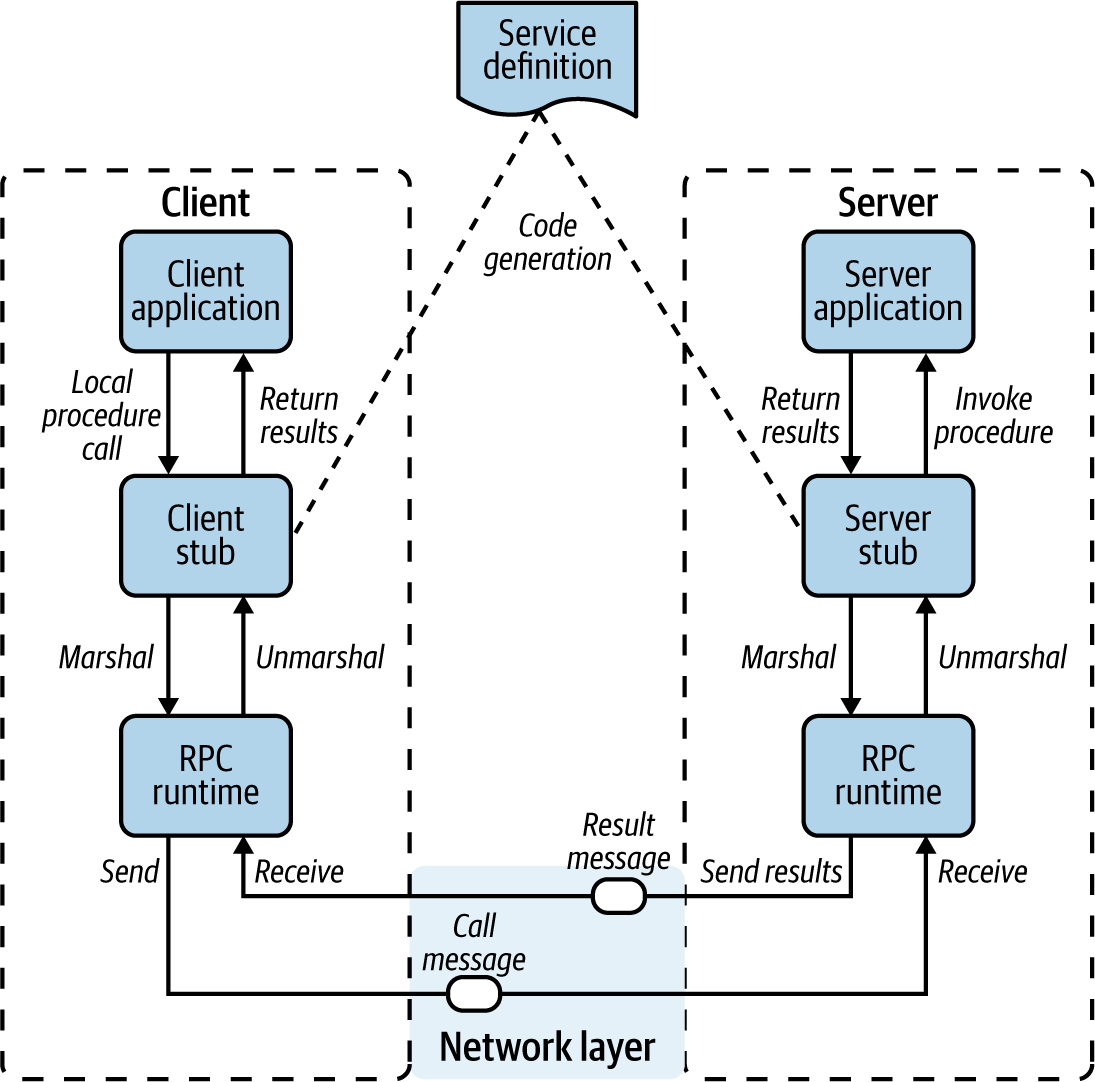
The *Remote Procedure Calls* (*RPC*) pattern is a synchronous communication pattern that enables distributed applications to invoke procedures of a remote application—just as if making a local procedure call. A given microservice can build a certain piece of business logic as a function and make it available for remote invocation by a consumer that resides in a separate process.

**How it works**

To understand how RPC works, imagine an online retail application that has a microservice for getting product details. We can expose that functionality as a remote procedure call so that external clients/consumers can invoke it as easily as making a local function call in their client application code. The underlying RPC framework handles all the complexities of the remote method invocation over the network and hides the details of the underlying network communication from the application developer. Implementing the RPC pattern allows you to use disparate technologies (different programming languages) to build your client and server applications.

Let’s look at [Figure 2-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#implementation_of_remote-id00201) to understand how RPC works. In RPC, the first thing you would do as the service developer is to come up with a *service definition* that outlines the details of the remote methods that you expose to the consumers, service name, name resolution, and data types that the service uses to exchange information. The language used to specify the service definition is an interface definition language (IDL).

RPC implementations allow you to use the service definition to generate client- and server-side code that handles the low-level protocol details of the RPC communication. These generated code components are called server or client *stubs*. As a consumer or as the microservice’s business logic developer, you don’t need to worry about the implementation details of the RPC technology; you can fully rely on the abstractions provided by the stubs and focus more on the business logic of your applications.



**Figure 2-2. Implementation of Remote Procedure Calls pattern**

When a client application wants to invoke a remote method by using RPC, the client application calls the client stub with the required parameters. This is a local function call, as the client stub resides within the client’s process or address space. The data types used inside the client application to communicate with the remote service are provided as part of the client stub itself. Then the client stub serializes (or marshals) the details of the remote service call into an encoded binary message, and the RPC runtime component passes the message to the underlying network transport layer to invoke the remote server application.

On the server side, the network transport layer passes the call message to the RPC runtime, which invokes the corresponding server stub (internally, it uses request-response messaging on top of Transmission Control Protocol, or TCP). The server stub deserializes (or unmarshals) the binary message and resolves the server procedure, mapping parameters and data types. Then it can call the corresponding server application’s remote procedure, where the server-side business logic resides.

Once the server’s business logic is executed, it returns the response data types to the server stub (returning the expected data type from the local function), which marshals it to the reply message. Then the server hands over the message to the RPC runtime, which sends it back to the client application over the network transport layer. Similar to the server side, the client stub unmarshals the return parameters, and execution returns to the client application.

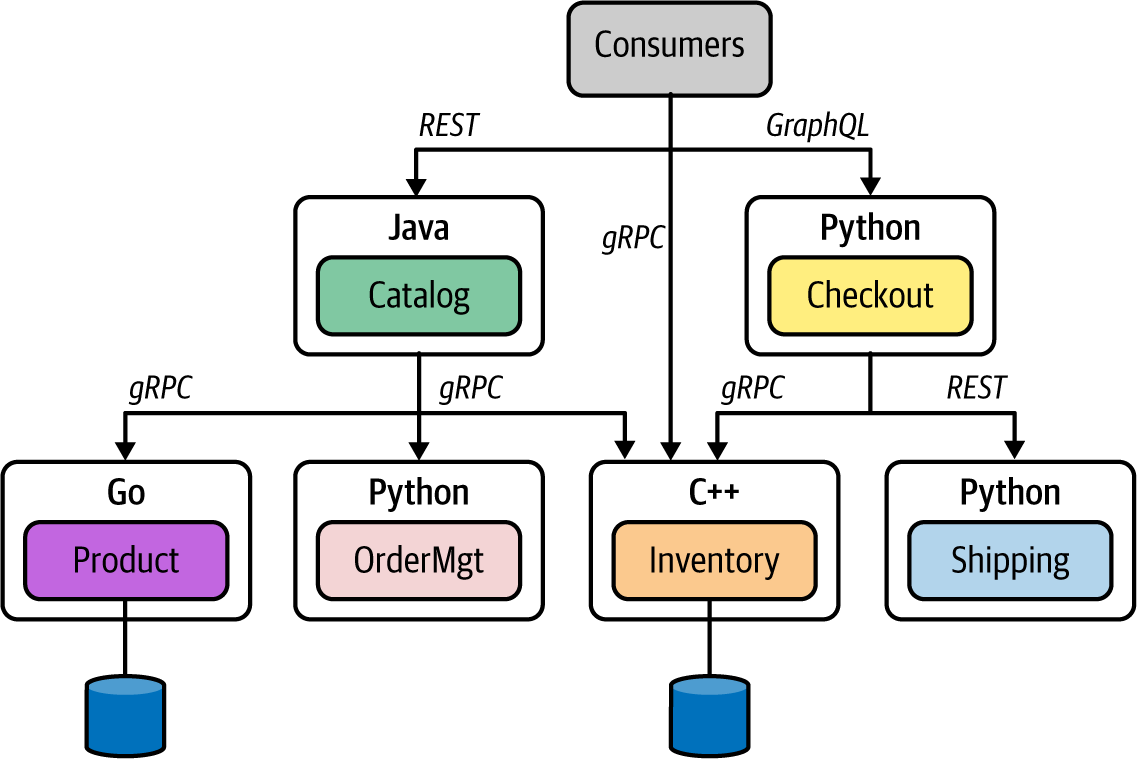
**How it’s used in practice**

In the context of cloud native applications, RPC can be used for most inter-microservices communications. Each microservice can be built as an RPC application, while each business capability is implemented as a remote procedure.

Numerous RPC technologies (such as Common Object Request Broker Architecture, or CORBA) have been used in distributed application development for decades. Most were directly built on top of TCP and were inherently complex. Therefore, the usage of such legacy RPC technologies has plummeted with the rise of RESTful services.

However, the RPC pattern has been reinvigorated with *gRPC*, a cloud native implementation of RPC that tries to overcome conventional limitations by using HTTP2 as the communication protocol (which is easier to interoperate with most of the existing communication components such as load balancers) and Protocol Buffers as the data serialization format (which is efficient and type safe).

[Figure 2-3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#using_grpc_for_microservice_communicati) illustrates a cloud native application built using multiple microservices implementing RPC-based interservice communication. Here we use gRPC as the RPC technology. RPC-based communication is often suitable for building the communication among services that are used internally.



**Figure 2-3. Using gRPC for microservice communication**

While no technical limitation prevents you from using gRPC for external communication (such as services exposed to the web), most RPC technologies lack the support required in web clients. gRPC is the most commonly used RPC technique in the context of cloud native applications. We’ll explore gRPC in detail later in this chapter.

**Considerations**

RPC is one of the most efficient and robust ways to build communication among microservices. Whenever you need to build synchronous communication among services, we recommend using cloud native RPC technologies such as gRPC. However, when the services need to be exposed to external consumers such as web and mobile applications, RPC may not be the best choice because these types of applications provide better support for other styles and data formats such as RESTful and JSON.

**Related patterns**

Here are some of the patterns that are related to Remote Procedure Calls:

*Service composition and API management patterns*

RPC is commonly used as a foundational pattern when realizing these other patterns (covered in Chapters [3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern) and [7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_and_consumption_patterns), respectively).

*Request-Response*

This is an alternative approach for RPC.

**Summary of Synchronous Messaging Patterns**

[Table 2-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#synchronous_messaging_patterns-id00279) summarizes when to use—and when not to use—the synchronous messaging patterns, as well as their benefits.

| **Pattern** | **When to use** | **When not to use** | **Benefits** |
| --- | --- | --- | --- |
| Request-Response | Services need real-time responses. Service contracts need to be flexible. To interoperate with many types of consumers. Services are exposed to external consumers. | Low-latency and high-throughput communication is required. Strict contract-first interactions are required. | The most interoperable and standard communication pattern for implementing services exposed to external as well as internal consumers. |
| Remote Procedure Calls | High-performance communication among services is critical. To enforce a strict contract-first approach for building services. Service business logic needs to be completely independent from the underlying wire protocol and its semantics. | Service interoperability with multiple application types such as web or mobile apps is required. You have to enable loose contracts and flexibility for consumers. | Suitable for efficient and type-safe service-to-service communication. |
| Table 2-1. Synchronous messaging patterns | | | |

**Asynchronous Messaging Patterns**

In asynchronous communication patterns, one microservice communicates with another microservice by sending data as messages without expecting a response. There may be no response at all, or the response may arrive asynchronously on a different channel (such as a separate queue).

With asynchronous messaging patterns, communication among microservices is facilitated by a third-party component known as a *message broker,* or *event broker.* This component receives messages from the source/producer microservice and sends them to the consumer. The consumers can consume the messages that they are interested in via the message broker.

**NOTE**

In this section, we discuss the foundational patterns related to asynchronous messaging. We dive into variants of asynchronous messaging patterns in detail in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).

When building cloud native applications, the broker applications provide a robust messaging infrastructure with minimal business logic. The business logic of the microservices should always be implemented at each producer and consumer microservice, but not at the message-broker level. Given that the broker is a messaging infrastructure that doesn’t have any business logic, it is common to use it as a centralized messaging platform (you don’t always need a dedicated broker runtime for each asynchronous messaging pattern implementation).

You can choose from multiple asynchronous communication patterns to build cloud native applications. Let’s begin our discussion with the Single-Receiver pattern.

**Single-Receiver Pattern**

In the *Single-Receiver pattern*, a given microservice delivers messages to exactly one target microservice, or to a system using a messaging infrastructure such as a message broker. The messages sent here are usually considered *commands* because the pattern ensures that the messages are delivered to a single consumer that is supposed to process them and perform an action.

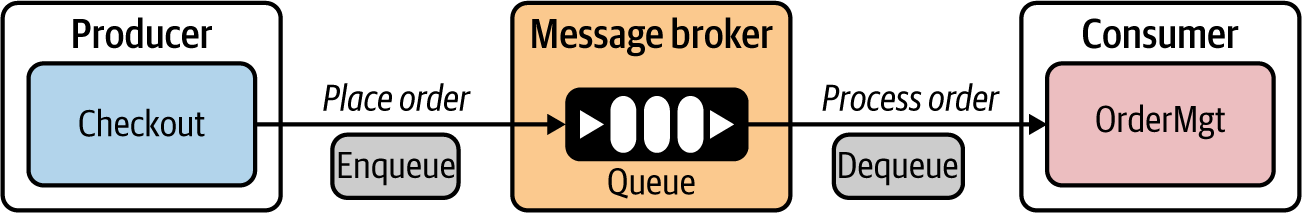
For example, when we have to process an order in an online retail system, we place an asynchronous message in a message broker queue, so that the Order-Processing service can process it and perform actions. Since it describes the information exchange between one producer and a single receiver, this pattern is also called *point-to-point asynchronous messaging.*

**How it works**

The Single-Receiver pattern is built by publishing messages to a queue in the message broker. One consumer service or system then consumes messages from that queue. The producer service is interested only in whether the message is delivered successfully to the queue; it doesn’t care when or whether the message is processed.

Since this pattern uses a queue to deliver messages from a producer to a consumer, it ensures the ordered delivery of the messages. The message broker offers the required message-delivery guarantees (such as at-least-once delivery) as part of the communication protocol it uses. Both producer and consumer can ensure the delivery of the message by using the semantics provided from the broker (such as acknowledgments after producing or receiving messages).

[Figure 2-4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#single_receiver_pattern_for_implementin) illustrates how two microservices of an online retail application communicate via the Single-Receiver pattern. In this example, the Checkout microservice publishes the order messages to a queue in the message broker. The OrderMgt service consumes those messages from the queue and performs the corresponding action.



**Figure 2-4. Single-Receiver pattern for implementing asynchronous message-based commands**

When producing messages, the Checkout service can ensure that the message has been delivered to the queue. At the receiver side, the OrderMgt service can send an acknowledgment after processing the message from the queue. The message broker’s queue ensures that the message is sent to only a single consumer. The Single-Receiver pattern is often used to implement guaranteed message-delivery use cases.

**How it’s used in practice**

The Single-Receiver pattern can be implemented using a wide range of message broker solutions. While the messaging semantics are similar, the implementation technology differs from broker to broker. Advanced Message Queuing Protocol (AMQP) is the most widely accepted protocol that facilitates queue-based single-consumer messaging. Other variants of the Single-Receiver pattern are discussed in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).

With AMQP, message brokers can support disparate (implemented in different programming languages) producer and consumer applications and establish the Single-Receiver pattern using queues. AMQP implementations such as RabbitMQ, Apache ActiveMQ, and Apache ActiveMQ Artemis are commonly used to implement this pattern. Also, fully cloud-based messaging solutions such as Microsoft Azure Service Bus support this pattern through the broker solution, which is offered as a cloud service.

**Considerations**

A queue-based Single-Receiver pattern is a commonly used messaging technique in which end-to-end guaranteed message delivery is required. The broker that we select to implement this pattern plays a major role in reliability, scalability, and performance of the cloud native application. Therefore, it’s important to select a broker technology that suits your exact requirements.

Most brokers that we can use to realize this pattern have roots in monolithic enterprise middleware. They tend to allow the developer to add quite a lot of business logic as part of message brokering (for example, routing, filtering, and content-aware delivery). Use them with caution in the context of cloud native applications. Our recommendation is to try to place business logic *outside* the broker and inside the microservices as much as possible.

**Related patterns**

Multiple patterns that are built on top of the base Single-Receiver pattern cater to various message-delivery semantics such as at-most-once and at-least-once delivery. In addition, variants such as the Fire and Forget pattern can be implemented without using a message broker. We explore this pattern in detail in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).

**Multiple-Receiver Pattern**

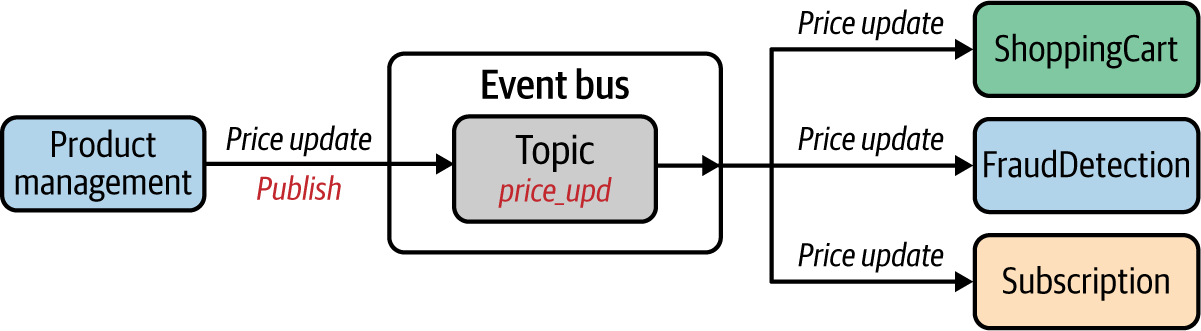
Single-consumer-based asynchronous messaging works when you have one consumer to consume messages that you publish to a message broker. What if you have to send the same message to multiple consumers who are interested in a particular event? This is where the *Multiple-Receiver*, or *Publisher-Subscriber*, pattern comes into the picture.

In our cloud native applications, we often need to build microservices that execute business logic upon the occurrence of certain events, or that notify one or more other microservices when a particular event occurs. In these interactions, we can use the Multiple-Receiver pattern.

**How it works**

In the Multiple-Receiver pattern, messages are delivered to more than one consumer microservice. We also use a message broker or an event bus to facilitate the asynchronous message delivery. One microservice publishes a message to a *topic* in the event bus, and one or more microservices can subscribe to a given topic. The message is asynchronously delivered to all the subscribers of that topic.

The Multiple-Receiver pattern is implemented using an event bus ([Figure 2-5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#multiple_receiver_pattern_for_implement)). Returning to our online retailer example, here we see that the price update of a particular item is published to the price update topic by the Product Management microservice. Several other microservices (such as ShoppingCart, FraudDetection, and Subscription) are interested in learning about events related to the price update topic.



**Figure 2-5. Multiple-Receiver pattern for implementing asynchronous message-based communication**

The event bus is responsible for handling the publisher and subscription requests and delivering the message to the corresponding subscribers. The delivery guarantees are not as strict as in the Single-Receiver pattern. In most cases, the event bus simply delivers messages to available subscribers. If the subscribers need to receive messages when they are offline for a long time, we can leverage durable subscription techniques that are implemented in certain event bus solutions. Keep in mind that supporting durable subscription increases the load on the event bus, as it needs to keep all the messages sent to those topics for the subscribers.

**How it’s used in practice**

The Multiple-Receiver pattern is implemented with the use of message broker solutions that support publisher-subscriber messaging. For example, conventional brokers (such as ActiveMQ, RabbitMQ, and Azure Service Bus) that support queue-based single-receiver models also support topic-based messaging. In addition, specific messaging technologies designed for event-based multiple-consumer scenarios such as Apache Kafka, Neural Autonomic Transport System (NATS), Amazon Simple Notification Service (SNS), and Azure Event Grid offer highly scalable event-driven messaging using the Multiple-Receiver pattern.

When it comes to a delivery mechanism, the Multiple-Receiver pattern is often implemented with support for persistent delivery, which means the events published by the producers are stored in a persistent store. However, when the events are published to subscribers, delivery of messages is not guaranteed by default to all subscribers, as some of them may not be reachable. Therefore, this pattern is used when delivery semantics such as at-least-once delivery are not required on the consumer side. However, certain brokers introduce such delivery guarantees with concepts such as *durable topics*: the broker logically persists an instance of each message for every durable consumer, since each durable consumer gets its own copy of the message.

As a general practice, if you want to send the same message to multiple parties with delivery semantics such as at-least-once delivery, then rather than using multiple consumers, you can use multiple queues for each consumer and publish messages as we did in the Single-Receiver pattern.

**Considerations**

The event bus component is generally used as a centralized runtime that is shared among multiple microservices. For example, all the implementations of multiple-consumer patterns may share the same broker instance. Therefore, it is important to keep the broker independent from the business logic (such as message routing based on certain properties of the message) and use it as a messaging infrastructure only. Then consumers will get more control over the consumption of the messages.

Also, as discussed earlier, we should bring in special message-delivery mechanisms such as durable subscription and durable topics only if the use case mandates it. At the event-bus level, we can include more-granular controls for the topic subscription and delivery of messages to the topics, and there is support for concepts such as hierarchical topics and routing rules. We explore those patterns in detail in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).

**Related patterns**

Event-driven architecture uses the Multiple-Receiver pattern as one of its core approaches. In [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns), we explore the various flavors of multiple-receiver messaging patterns.

**Asynchronous Request-Reply Pattern**

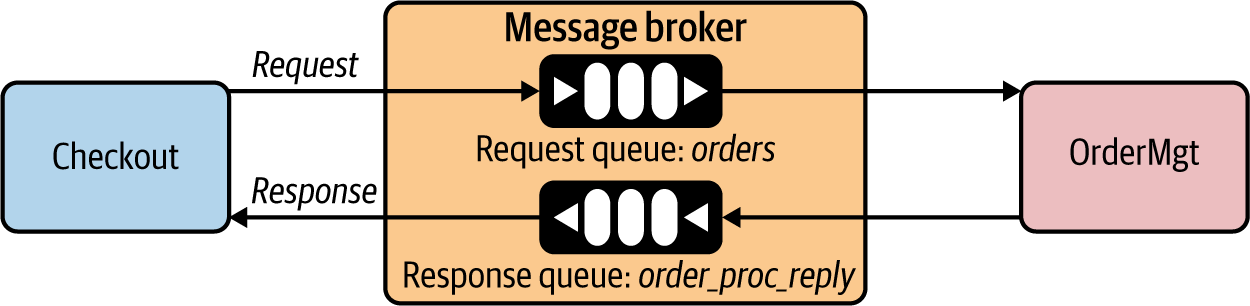
What we have discussed so far regarding asynchronous communication has mostly been about one-way messaging: we send data without expecting any response in return. But in some cases, producers need to send messages to a consumer via a broker and receive a reply from the consumer via the broker on a different communication channel. This is where the *Asynchronous Request-Reply pattern* comes into play.

**How it works**

In the Asynchronous Request-Reply pattern, we follow the same messaging model used in the Single-Receiver pattern: the producer microservice publishes messages to a queue in a message broker, and then the producer consumes that message from the queue. However, the message contains metadata specifying that it requires a reply, the location where the reply should be sent, and how to correlate the reply. The producer uses that information to send the reply back to the producer via a completely different channel established through a separate queue in the message broker.

In [Figure 2-6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#asynchronous_request_reply_pattern_impl), a producer microservice—Checkout—sends a request to a *request queue* in the message broker. Since we intend to process a reply, the message sent from the producer may contain a correlation ID and reply channel information. When a message is successfully sent to the queue, the producer ensures only that the message is successfully produced to the queue but doesn’t worry about the reply. The reply is handled by a completely different flow in the producer: the producer initializes a separate subscription to a predefined *response queue* (also known as *callback queue*) that resides in the message broker.

On the consumer side, when the message is consumed, it obtains the reply channel information and correlation ID and places the reply on the response, or callback, queue. As we listen for it from the producer side, we can process the reply message asynchronously.



**Figure 2-6. Asynchronous Request-Reply pattern implemented using a message broker**

In this example, the Checkout service sends an order-processing request to the request queue with an order ID and reply queue name (order\_proc\_reply). Then the consumer OrderMgt service processes it and delivers the response to the response queue.

**How it’s used in practice**

The Asynchronous Request-Reply pattern is not an alternative to the Request-Response pattern you learned about previously. This pattern serves a specific purpose; when you need to send back a reply that contains business data to a given asynchronous message, you need to use this pattern. Therefore, most of the messaging solutions that support the Single-Receiver pattern facilitate the Asynchronous Request-Reply pattern as well.

Broker implementations such as RabbitMQ, ActiveMQ, and Azure Service Bus support the Asynchronous Request-Reply pattern. We further discuss some of the other patterns that you can build on top of this one in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).

**Considerations**

Because of its similarities with the Request-Response pattern, you may think the Asynchronous Request-Reply pattern is a better and more reliable alternative to synchronous request-response messaging, because we can use queues to ensure persistent messaging between the parties. However, keep in mind that these two patterns are used for completely different use cases.

The Asynchronous Request-Response pattern is a combination of two one-way messages. However, the performance implication due to using a queue for both request and response, and the overhead of correlating messages, need to be taken into account when applying this pattern. In the real world, we don’t see this pattern used as frequently as the other asynchronous communication patterns that we’ve discussed.

**Related patterns**

This pattern is essentially a combination of two asynchronous single-receiver patterns built in opposite directions. So most of the delivery semantics that you apply for the Single-Receiver pattern can be applied for this pattern as well.

**Summary of Asynchronous Messaging Patterns**

[Table 2-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#asynchronous_messaging_patterns) summarizes the asynchronous messaging patterns, detailing when to use them and when not to.

| **Pattern** | **When to use** | **When not to use** |
| --- | --- | --- |
| Single-Receiver | One microservice sends an asynchronous command to another microservice. For ordered message delivery. For guaranteed message delivery. | Efficient data transfer is required without delivery semantics such as at-least-once. |
| Multiple-Receiver | More than one consumer is interested in the same message/event. | Usually not suitable when you need guaranteed message delivery. |
| Asynchronous Request-Reply | For asynchronous messaging scenarios in which correlation is required between a request and a reply. | Shouldn’t be used as a reliable messaging alternative to synchronous request-response patterns. |
| Table 2-2. Asynchronous messaging patterns | | |

**Service Definition Patterns**

When building communication among cloud native applications, one of the most important aspects to consider is the *service definition*—how you define your microservice interface to its consumers. Service definition techniques, and how we use them, differ. Let’s first focus our discussion on using service definitions in synchronous communication.

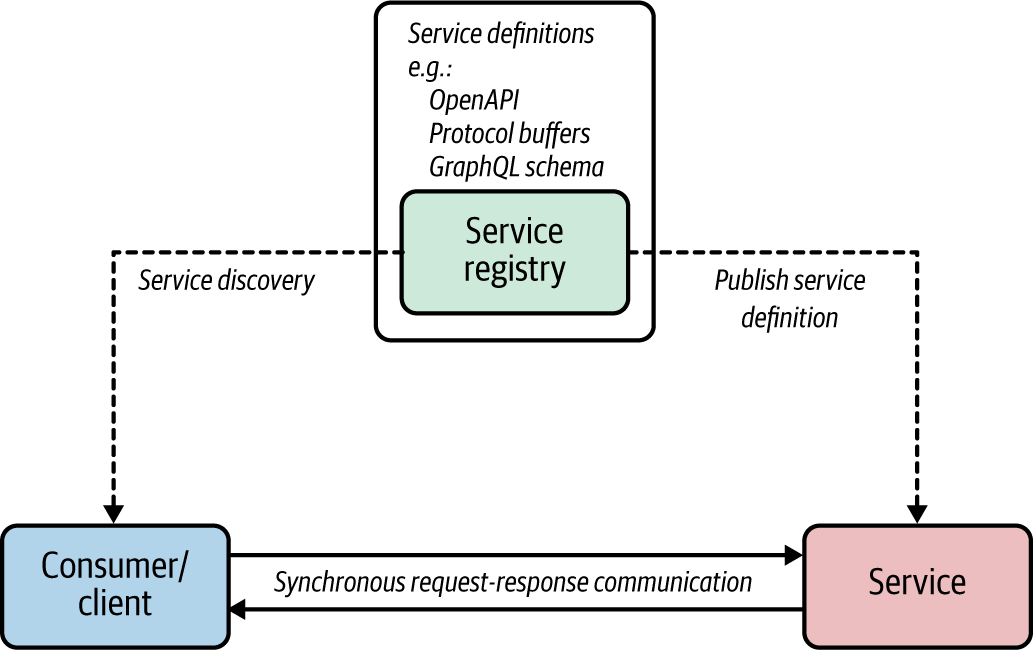
**Service Definitions in Synchronous Communication**

When we build synchronous services, we can publish the service definitions to a central location known as the *service registry*. This is more or less a metadata repository that the other microservices and developers can interact with. In defining the service interface, you can choose from a wide range of technologies, depending on the communication protocol you’re using for synchronous communication (for example, RESTful with HTTP, gRPC, and so on).

**How it works**

The service definition is a way of declaring how a given service can be consumed by consumers/clients. It is something that will be shared with consumers prior to establishing the communication channel among the services where synchronous communication takes place.

As illustrated in [Figure 2-7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#using_a_service_registry_for_storing_se), microservices can publish their service definitions to a service registry (or they can be published manually by the microservice owners). Then the consumers of those microservices can connect to the service registry and obtain the service definitions (programmatically or manually by the developers). This step is known as *service discovery.*



**Figure 2-7. Using a service registry for storing service definitions and discovering them**

The service definitions that the consumers obtain can be used to build the client application or generate the required client libraries to communicate with the server. The service definition contains the interfaces that a given service offers, as well as message formats and schema for data types exchanged between the client and microservices.

A service registry holds the service definitions and may offer additional capabilities such as user ratings, reviews, and support for various API collaboration requirements. We explore the Service Registry and Discovery pattern in detail in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern).

**How it’s used in practice**

A service registry is often implemented as a metadata repository with an API to manage service definitions and other metadata. Some tools that can do this are Consul, etcd, and Apache ZooKeeper. In most deployments, we can run the service registry as a centralized component. The Service Registry and Discovery pattern is discussed in detail in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern), but for the context of service definition patterns, we can just assume it’s a metadata repository for putting and retrieving service metadata.

The service definition techniques differ from one protocol to another. For instance, RESTful services use OpenAPI, gRPC uses Protocol Buffers, and GraphQL uses GraphQL schemas to define service interfaces. While some protocols (gRPC) mandate the use of a service definition, others don’t require a service definition at all (REST). However, in the context of cloud native applications, since we are interacting with numerous microservices, it is always a good practice to use a service definition in each and every service that we develop, and to ensure that it is discoverable through a mechanism such as a service registry.

**Service definitions for RESTful services with OpenAPI and Consul**

You can define the business interface of a microservice in your cloud native application by using service definition specifications such as OpenAPI. You can then store that definition in a service registry such as Consul, so that the consumers of your service can obtain the metadata required to access the service. By using the OpenAPI definition, consumers can generate code, refer to documentation, understand service-level agreements, obtain supported security schemas, and so on. Service registries like Consul provide a uniform way to access the service metadata and act as the service catalog for obtaining information on any service that is part of your organization.

**Considerations**

The service definition and how to share that definition with the rest of the organization is tightly coupled to the overall governance aspects of building cloud native applications. We usually store service definitions in a central service registry so all the consumers of the microservices that we build can discover them. Therefore, the service registry should be capable of handling disparate sets of service definitions, such as the OpenAPI specification, gRPC service definitions, and GraphQL schemas.

With the introduction of service definitions, the build and the development life cycle of your microservice also changes. Often when you have a service definition, you can code-generate or validate your microservice implementation against the service definition. This ensures that every release of your microservice complies with the advertised service definition.

**Related patterns**

Service definition patterns are closely related to the Service Registry and Discovery pattern explained in detail in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern). In addition, we can apply this pattern along with synchronous Request-Response and Remote Procedure Calls covered previously in this chapter, and the API management patterns covered in [Chapter 7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_and_consumption_patterns).

**Service Definition in Asynchronous Communication**

In asynchronous communication, messages exchanged between producers and consumers contain structured data that is being serialized or deserialized using a *schema* that defines and validates the data exchanged between the parties. Since the communication happens asynchronously through a message broker or an event bus, the microservices that do the producing and consuming of messages should use a common schema. Similar to service definitions in a synchronous messaging scenario, the producer and consumer microservices have to use a central metadata registry to store the schemas.

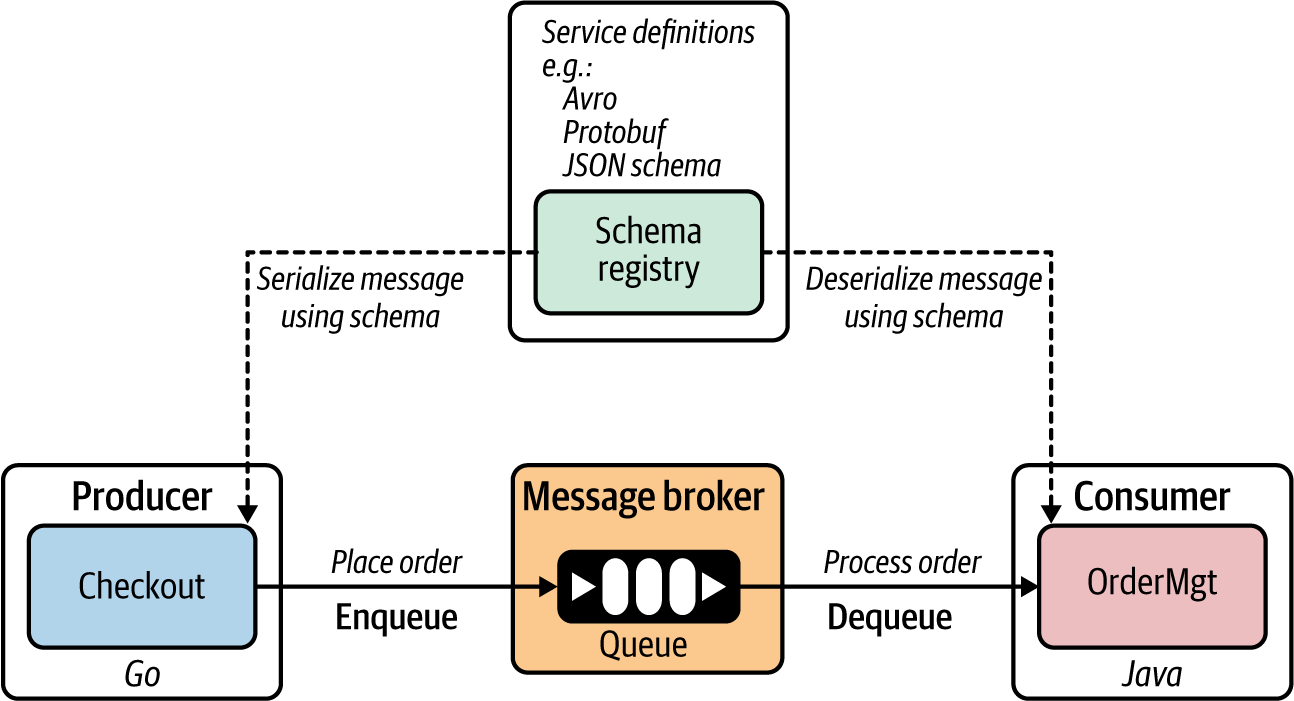
**How it works**

Asynchronous communication patterns don’t create any coupling between the producer and consumers. So we tend to think that we can publish any arbitrary message to the interested topic and expect the consumer to process it somehow. However, this is far from the reality.

To build robust communication among microservices that use asynchronous communication, you need to use a service definition that specifies the asynchronous messaging contract (which includes the type definitions of the messages exchanged between the producers and consumers). Often this contract contains the schema definition of the messages exchanged between producers and consumers.

When two microservices use asynchronous message-driven communication, the producer can validate the message against a schema (residing in the schema registry) while publishing the message to a queue or topic in a message broker. In our online retail application depicted in [Figure 2-8](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#the_service_registry_contains_the_servi), the Checkout service, publishing a message to a queue, will adhere to the service definition that contains the order schema in the registry when it serializes the messages. If the schema validation fails, the producer can’t publish a message to the broker. Under the hood, the producer connects to the schema registry to retrieve the service definition with the schema and validate it.

The consumers follow the same pattern when they deserialize the message so that they read the message based on the schema provided in the registry. In this example, the OrderMgt service can use the order schema from the service registry.



**Figure 2-8. The service registry contains the service definitions and schemas for messages exchanged between producers and consumers.**

When there’s a mismatch of the data either being sent or consumed (for example, the producer type is different from the type that is expected by the consumer), it can be detected at the producer or consumer side during publishing or consumption of messages.

By using schema-based data serialization and deserialization, asynchronous message-based communication can drastically reduce the amount of metadata related to type information and field names that you have to pass along with every message.

**How it’s used in practice**

In most early implementations of asynchronous communication, schemas and schema registries were not commonly used. However, with the proliferation of asynchronous communication among microservices, the necessity of having a clearly defined contract for messages exchanged between producers and consumers has increased. Therefore, many message broker solutions offer first-class support for a schema registry as part of producing and consuming messages.

Similar to the service definitions that we use in synchronous communication, we can use a central metadata repository such as a service registry, and for the schema definition techniques we can use Apache Avro, Protocol Buffers, or JSON schemas. Depending on the type of broker you use, the schema definition technique may vary. For example, Kafka supports Avro, and Azure uses Azure Schema Registry in its Event Hubs messaging service. Emerging asynchronous service definition technologies such as [AsyncAPI](https://www.asyncapi.com/" \t "_blank) can be used to specify the entire service contract rather than just the schema for the messages.

For applications that use event-driven architecture, the service definitions can leverage event-describing specifications, such as [CloudEvents](https://cloudevents.io/" \t "_blank). In this way, we can simplify the event declaration and delivery across disparate microservices and other systems. We discuss these techniques in detail in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).

**Schema definitions with Kafka Schema Registry**

Kafka supports the integration of message consumers and producers with the Kafka Schema Registry, in which you can store and retrieve your schema definitions for messages exchanged between producers and consumers in asynchronous messaging. You can store Avro, JSON, and Protocol Buffers schemas in the Kafka Schema Registry, and from your producers and consumer applications you can validate the schema compliance while producing or consuming messages.

The Schema Registry stores a versioned history of all schemas and provides serializers that plug into Kafka clients. These clients handle schema storage and retrieval for Kafka messages that are sent in any of the supported formats.

Cloud messaging services such as Amazon Kinesis and Azure Event Hubs also support this pattern, with seamless integration among their respective schema registry services.

**Service definitions with AsyncAPI**

Schema registries provide only the schema for messages exchanged between producers and consumers. However, these registries don’t specify any contract details related to publishing or consuming messages. AsyncAPI tries to standardize the definition of the service contracts for producers and consumers in asynchronous messaging. Although it has not yet been widely adopted by the community, it has some promising characteristics of becoming the standard service definition for asynchronous messaging architectures.

**Considerations**

Most asynchronous messaging is implemented without using schema-based serialization and deserialization. This often leads to inconsistencies and data-type mismatching between producers and consumers. In addition, the amount of metadata that needs to be sent along with the message increases the message size, which slows the performance of asynchronous communication. Therefore, adopting schemas in asynchronous messaging is vital for ensuring the reliability and safety of cloud native applications.

We discussed schema validation on both the producer and consumer side. This inherently introduces performance overhead, as each message needs to go through a validation process. Also, pulling the schema from the registry may require a caching mechanism to avoid performance bottlenecks.

**Related patterns**

Service definition patterns are closely related to the Service Registry and Discovery pattern detailed in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern). We can apply this pattern along with Asynchronous Request-Reply and Remote Procedure Calls (covered previously in this chapter) as well as the API management patterns (covered in [Chapter 7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_and_consumption_patterns)).

Now that we have covered all the foundational communication patterns for building cloud native applications, let’s wrap up the chapter by discussing technologies that enable the patterns we’ve discussed.

**Technologies to Implement Synchronous Messaging Patterns**

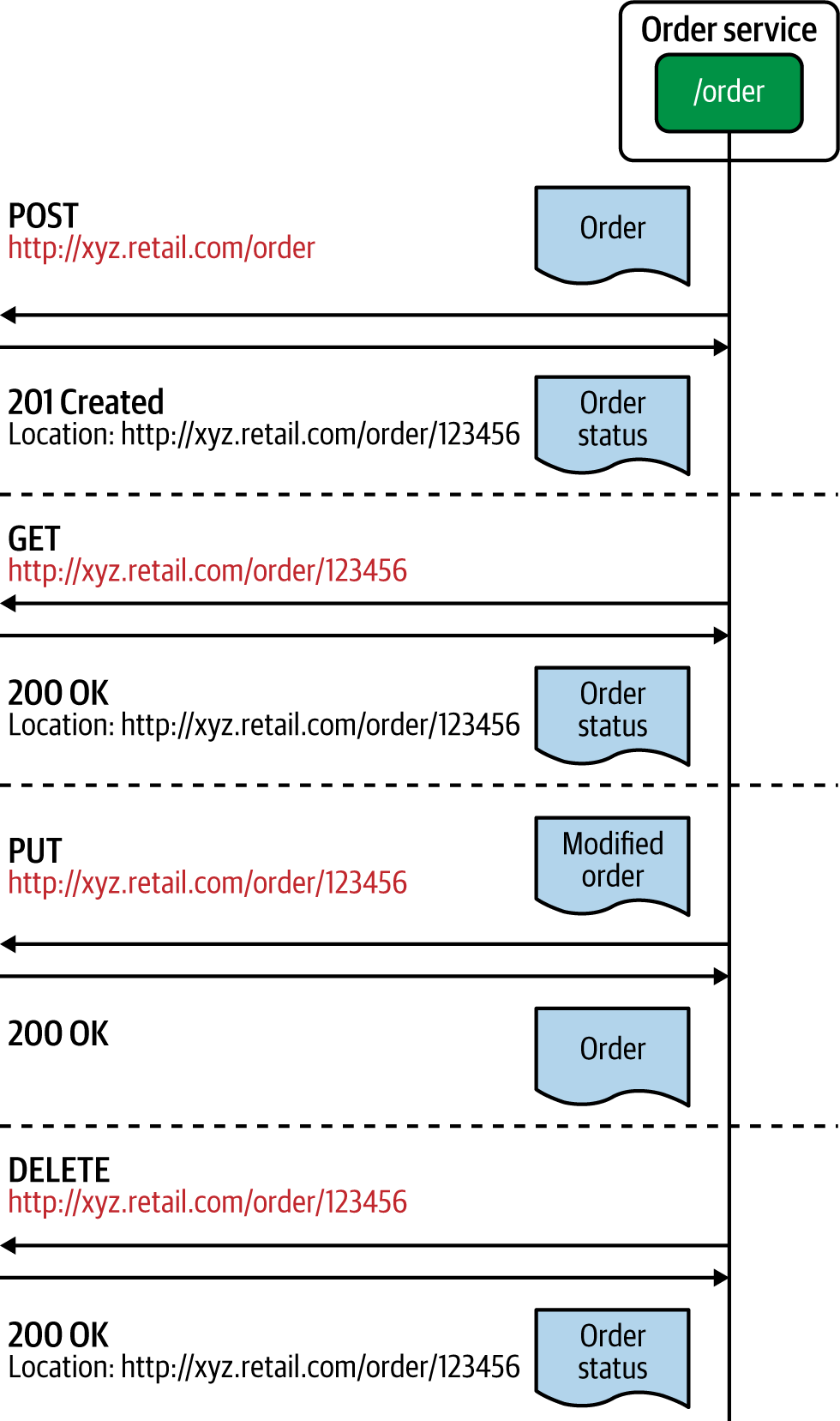
Most implementations of the Request-Response pattern leverage protocols such as HTTP while using different data representations and exchange techniques to transfer data between microservices.

**RESTful Services**

*RESTful services* are one of the most popular ways to build a cloud native application’s microservices using the Request-Response pattern. They are built on top of the REST architectural style. The REST model uses a navigational scheme to represent objects and services over a network. These are known as *resources*—objects with a type, associated data, relationships to other resources, and a set of methods that operate on them (for example, an order is a resource in the context of an online retail application where you can perform multiple actions). A client can access (request) the resource by using the unique URI, and a representation of the resource is returned (as the response).

REST doesn’t depend on any of the implementation protocols, but the most common implementation is HTTP. When accessing RESTful resources with HTTP, the URI of the resource serves as the resource identifier, and GET, PUT, DELETE, POST, and HEAD are the standard HTTP operations to be performed on that resource. Therefore, all the requests are sent in the form of those HTTP operations.

[Figure 2-9](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#an_example_restful_service_and_its_inte) shows a real-world implementation of a RESTful service: an Order service in an online retail application.



**Figure 2-9. An example RESTful service and its interactions**

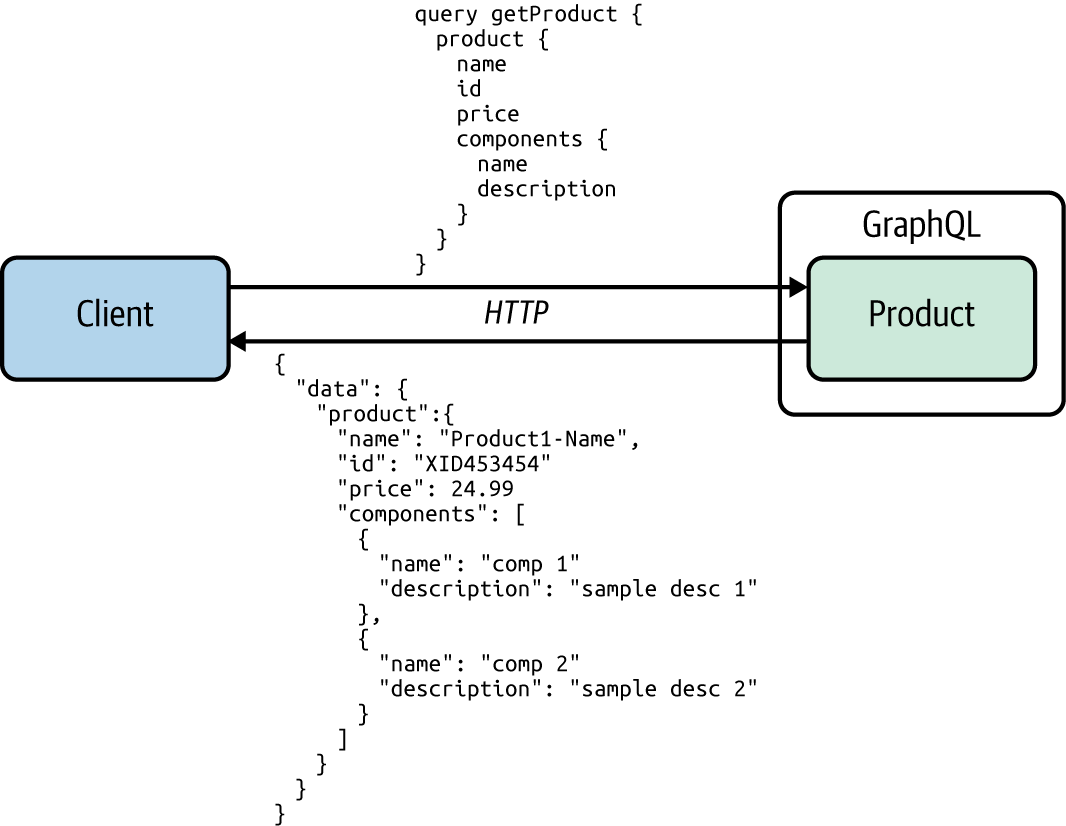
In this example, we have represented an Order as a resource, and all requests sent in the form of HTTP operations are executed against that entity. The format of the request and response data is an implementation detail that’s agnostic of the REST architectural style. Often JSON, Extensible Markup Language (XML), and other text-based data formats are used. For example, we can send an order-creation request by using an HTTP POST message to the order resource located at a given URL. Similarly, we can retrieve, update, or delete the order resource by sending HTTP GET, PUT, and DELETE requests, respectively.

**GraphQL**

*GraphQL* is another technology that is becoming popular for building interprocess communication using the Request-Response pattern. Unlike RESTful services, GraphQL is based on the concept of sending a query as a request to the microservice. The query represents the data that the client is interested in, and the microservice’s logic fulfills those queries with the existing data and business logic.

GraphQL allows clients to determine which data they want, how they want it, and in what format. This is different from RESTful services, where the client doesn’t have control over the response data that it receives. GraphQL primarily uses *queries*, *mutations*, and *subscriptions* as the main interaction styles with consumers and services. With a query, the client can request the data it needs from the server, while mutations are mostly used to modify data on the server. GraphQL also supports other messaging styles such as asynchronous event-driven communication via subscriptions.

[Figure 2-10](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#an_example_graphql_service_and_its_inte) shows how a real GraphQL microservice works. Here, the client sends the request to the microservice via the GraphQL queries and decides what data it wants in the response. If a given service has multiple entities, you don’t have to send explicit requests to retrieve them, but a single GraphQL query can do that for you. GraphQL requests are served over HTTP under the hood. A standard GraphQL request is sent as an HTTP POST request; it should use the application/JSON content type and include a JSON-encoded body (queries can be sent as an HTTP GET request with query parameters as well).



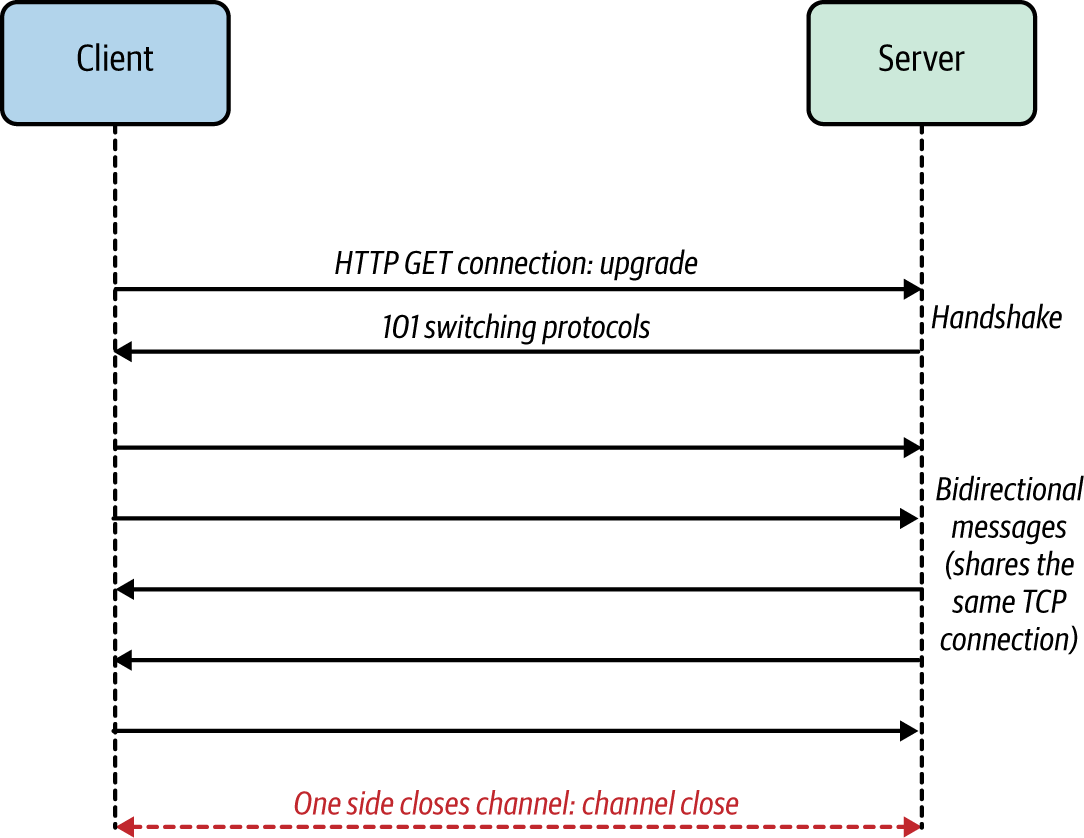
**Figure 2-10. An example GraphQL service and its interactions**

GraphQL is commonly used for external-facing microservices or APIs that are directly exposed to consumers (clients such as mobile applications), and for clients that need more control over the data that they consume from the server. In addition to Request-Response, GraphQL supports other messaging patterns such as publisher-subscriber messaging.

In comparison to REST, GraphQL offers an efficient way to fetch data without over-fetching (retrieving redundant data not required for the consumer) or under-fetching (retrieving only a portion of required data, which results in subsequent requests to fetch the remaining data). With GraphQL, the consumer can fetch the exact data needed in a single request. GraphQL provides other advantages including validation and type checking, detailed error handling, and backward-compatible versioning.

**WebSocket**

One technology used to implement the synchronous Request-Response pattern is the *WebSocket protocol*, which can simply be introduced as TCP over the web. However, it is more powerful than just a simple request-response protocol, as it supports full duplex (allows communication in both directions) and asynchronous messaging once the connection is established. WebSocket uses a single TCP connection for traffic in both directions and uses HTTP as the initial handshaking protocol ([Figure 2-11](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#an_example_of_websocket_communication)). Therefore, it can work with the existing infrastructure and works on top of HTTP for the initial handshake. It acts like raw TCP sockets afterward.

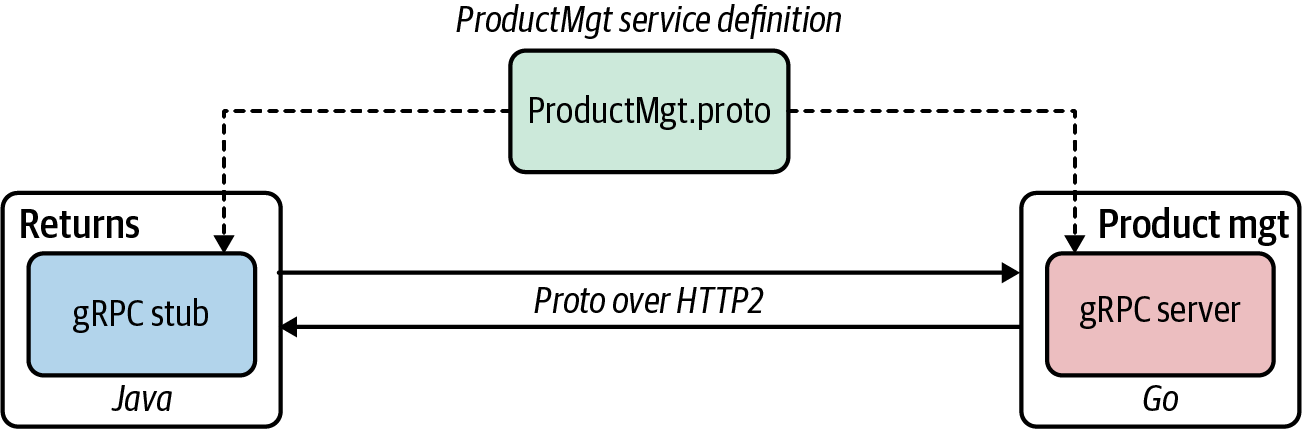


**Figure 2-11. An example of WebSocket communication**

WebSocket doesn’t mandate a specific data serialization format for messages, so you can choose any message format you like. WebSocket is useful for building microservice applications that use synchronous request-response communication as well as duplex messaging between the server and the client applications.

**gRPC**

*gRPC* is an RPC-based communication technology. It makes connecting distributed heterogeneous microservices as easy as making a local function call. gRPC is designed to be an internet-scale interprocess communication technology that can overcome most of the shortcomings of conventional RPC technologies. It uses an efficient binary data serialization format, Protocol Buffers, to marshal and unmarshal data exchanged between the client and server applications. Also gRPC is implemented on top of HTTP2, which makes it an interoperable and efficient RPC technology. It offers language plug-ins for all the popular programming languages so you can build polyglot cloud native applications. [Figure 2-12](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#an_example_of_microservices_communicati) illustrates the key components of a gRPC-based communication between two microservices.



**Figure 2-12. An example of microservices communication using gRPC**

When you develop a gRPC application, the first thing to do is define a service interface. As we’ve discussed, a service interface definition indicates how your service can be consumed by consumers, which methods allow consumers to call remotely, and which method parameters and message formats are used when invoking those methods. gRPC uses Protocol Buffers to specify the service interface definitions of gRPC-based microservices.

Using the service definition, you can generate the server and client stubs. As we covered earlier, the methods that you specify in the service interface definition can be remotely invoked by the client side as easily as making a local function call. The underlying gRPC framework handles all the complexities normally associated with enforcing strict service contracts, data serialization, network communication, authentication, access control, and observability.

Apache Thrift, another RPC framework similar to gRPC, uses its own interface definition language and offers support for a wide range of programming languages. gRPC is more opinionated than Thrift and offers first-class support for HTTP2. gRPC implementations on HTTP2 leverage the protocol capabilities to achieve efficiency and support for messaging patterns such as streaming.

**Summary of Synchronous Messaging Technologies**

[Table 2-3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#synchronous_messaging_technologies) compares the synchronous message technologies that we’ve discussed.

| **Synchronous messaging technology** | **When to use** |
| --- | --- |
| RESTful services | The business use case fits the resource-oriented model (in which you can represent business entities and functionalities as HTTP resources and operations). The service needs to interoperate with disparate sets of clients (web clients, mobile clients). The service needs to negotiate to support various content types (JSON, CSV, XML) based on client requests. You need human-readable text-based message formats. |
| GraphQL | Clients want to determine the data they want, how they want it, and the data format. You want a well-defined yet flexible schema for interservice communication. You want to reduce the number of service calls needed to retrieve business data from a service. |
| gRPC | You require low-latency and high-throughput interservice communication. You want type-safe and robust data exchange between microservices. The client or the server application needs to build a streaming business operation. |
| WebSocket | You have to implement full-duplex efficient messaging between services using your own data formats. |
| Table 2-3. Synchronous messaging technologies | |

**Technologies to Implement Asynchronous Messaging Patterns**

Let’s discuss some of the most commonly used technologies for implementing asynchronous messaging patterns in this section. We’ll begin our discussion with an overview of AMQP.

**AMQP**

The *Advanced Message Queuing Protocol* (*AMQP*) is the most commonly used protocol for implementing the Single-Receiver messaging pattern. AMQP facilitates reliable communication of asynchronous messaging among producers, brokers, and consumers. It ensures rapid and reliable message delivery, and message acknowledgments for both producing and consuming services. Message acknowledgments can be used on both the producer and consumer side. When a producer delivers a message to a queue, the broker sends acknowledgments, and when a message is delivered to a consumer, the consumer notifies the broker, either automatically or when the application code decides to do so. In message acknowledgments mode, the broker will completely remove a message from a queue only when it receives a notification for that message (or group of messages).

AMQP is a language-agnostic protocol, so you can use it to build asynchronous message-based communication among microservices or applications that use disparate programming languages and frameworks.

We discuss several other patterns in which AMQP becomes useful and some message brokers that implement this protocol in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).

**Kafka**

*Apache Kafka* is a distributed open source event bus/broker solution built on the concept of maintaining messages/events as a distributed commit log. The messages in Kafka are stored durably, in order, and can be read deterministically by multiple consumers at their own speed. Kafka is designed to be a highly scalable and distributed event broker. As such, it tries to give more control of messaging to producers and consumers while providing a robust, reliable, efficient, and scalable infrastructure. Kafka is a good choice when building a cloud native application that uses asynchronous communication patterns with heavy business logic that resides within the service logic itself.

As Kafka does not remove the events from the log upon delivery, it enables the replay of events. It uses an event *sequence number* to enable consumers to track their position in the stream to allow for selective replay. Kafka does not support protocols such as AMQP, Streaming Text Oriented Messaging Protocol (STOMP), or Message Queuing Telemetry Transport (MQTT), and it does not provide event queue semantics. Still, it is widely adopted for building event-driven architectures because of its high-performance characteristics and event-delivery guarantees. Kafka also provides extensions for implementing stream-processing systems.

**NATS**

[NATS](https://nats.io/) is a simple, open source messaging infrastructure specifically built for cloud native applications. Its key objectives are ease of use for developers and operators, high performance, high availability, lightweight messaging, and support for polyglot applications. NATS facilitates message-delivery semantics such as at-most-once and at-least-once for the Single-Receiver and Multiple-Receiver patterns. Like Kafka, it also uses logs for storing events, uses event sequence numbers to track events, and provides the ability to replay.

It does not have support for protocols such as AMQP, STOMP, or MQTT. But because of its lightweight nature, scalability, and native integration with Docker, Kubernetes, service meshes, and other cloud native technologies, it has become one of the first message brokers that is truly cloud native. It also supports event streaming, and command-and-control management for Internet of Things (IoT) and edge systems.

We explore a wide range of other asynchronous messaging technologies that are useful in building event-driven cloud native applications in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).

**Testing**

A cloud native application consists of multiple collaborating microservices that use different communication patterns. Therefore, testing strategies for these applications depend heavily on the underlying communication patterns that we use.

For synchronous communication, we can often isolate a given service and verify the capabilities by running tests against that service interface. As part of the tests, we send test requests to the service and verify that we get the expected responses, throughput, error messages, and so on. If the service calls multiple other services (these patterns are discussed in detail in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern)) and systems, we can still verify the composite business capability by running tests against that composite service.

However, for asynchronous communication, the testing strategy needs to be drastically changed. Unlike with synchronous messaging, we cannot test the service just by sending a request and evaluating the response. Even the simplest producer and consumer asynchronous messaging scenario requires messages to be published to a message queue (or topics) in the broker; then the consumer subscribes to it, and then it processes the message at the consumer side. Because all these entities are decoupled, it’s really hard to verify the end-to-end asynchronous messaging functionality with unit tests. We need to break down the scenario so that the producer service ensures that the required message is published to the broker and verifies it by consuming the message it has produced to the broker in the tests. The consumer can test its business logic by consuming identical messages from the broker.

While this process verifies the functionality of the producer and consumer to a certain extent, to ensure the proper execution of the end-to-end use case, we need to test producer, broker, and consumer all at once in a single deployment. This is essentially an integration test that scaffolds up the testing environment with the required configuration for producer and consumer services. We can automate such tests by using Docker Compose or a Kubernetes deployment.

**Security**

Implementing communication patterns in a secure way is a key requirement of any application that you build. Depending on the communication pattern that you use, the way you secure a cloud native application may differ. For synchronous messaging, we can use Transport Layer Security (TLS) to secure the communication channels between each microservice. This is applicable to any synchronous messaging technique such as RESTful services, gRPC, or GraphQL. Synchronous messaging communication patterns can be used along with other identity and access management patterns such as delegated authorization with OAuth 2.0 and federated identity with JSON Web Token (JWT). The details of these technologies are beyond the scope of this book, but we highly recommend *Microservices for the Enterprise* by Kasun Indrasiri and Prabath Siriwardena (Apress) for more details.

In asynchronous messaging, we don’t have a notion of service-to-service security as we have in synchronous communication. Rather, we secure the connectivity between the producer and the broker, as well as the broker and the consumer. To secure the communication by authenticating producers and consumers, we can expose broker endpoints via TLS, so that both producers and consumers use secured channels for producing and consuming messages. And we can implement authorization to access the broker by using technologies such as access control lists (ACLs), which are supported in most message brokers. We explore event-driven and streaming-architecture-specific security considerations in Chapters [5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns) and [6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#stream_processing_patterns-id00204).

**Observability and Monitoring**

Observability of cloud native applications is more or less independent of the type of communication technology we use. We simply have to use or import the agents or plug-ins related to metrics, tracing, logging, and service visualization in our application code. The underlying observability tools will take care of collecting, analyzing, and presenting the data related to observability.

For synchronous communication, all the technologies that we discussed in this chapter offer first-class support for integrating with observability tools. Therefore, minimal work is required from the cloud native application developers to build observable services. For asynchronous communication, certain aspects of observability such as tracing may require additional input from the application (such as correlation IDs) to determine the flow of messages, as it involves intermediaries such as message brokers. The low-level details of how observability and monitoring are implemented for cloud native applications are beyond the scope of this book.

**DevOps**

When it comes to automating and integrating the processes between software development and IT operations, most platforms and tools seamlessly work with the foundational communication patterns covered in this chapter. In particular, all the synchronous communication patterns integrate flawlessly with platforms such as Kubernetes as well as the cloud services offered from the main cloud providers.

For asynchronous communication patterns, the deployment style, workload state, and scaling and high-availability needs may be drastically different from that of synchronous communication. For example, scaling a broker deployment in a Kubernetes cluster requires extra effort (such as setting up a broker deployment using stateful storage and so on) compared to running an application that uses synchronous messaging. Most asynchronous messaging solutions offer some abstractions (for example, Kubernetes operators for [Kafka](https://strimzi.io/) and [RabbitMQ](https://oreil.ly/L44RJ" \t "_blank)) to simplify the DevOps tasks or offer the solution as a cloud service so that most of the DevOps-related work is included as part of the cloud service itself.

All the generic DevOps best practices can be applied when we build cloud native applications. We highly recommend Martin Fowler’s online [Software Delivery Guide](https://oreil.ly/NI0c6), which identifies various delivery strategies and DevOps patterns that you can apply to cloud native applications.

**Summary**

Cloud native communication patterns are applied when microservices of a cloud native application communicate with one another and with external systems. With the proliferation of microservices and increasing business requirements, we need to use a wide range of communication patterns when building cloud native applications. The two main categories of these patterns are synchronous and asynchronous.

In synchronous patterns, the client or consumer microservice/application expects a timely response from the microservice that it invokes. Most of the commonly used patterns such as Request-Response and Remote Procedure Calls fall under this category.

Asynchronous communication is all about delivering messages between the producers and consumers by using an intermediate messaging infrastructure called a message broker or event hub. The Single-Receiver and Multiple-Receiver patterns are the two main types within this category. The Single-Receiver pattern has a queue-based message delivery mechanism that is commonly used for ordered and reliable delivery of messages between a producer and a single consumer. The Multiple-Receiver pattern enables more than one consumer to receive the same message.

The service definition of a cloud native application also plays an important role in establishing which communication patterns you use. While service definition techniques may drastically differ from one protocol to another, the way we use the schemas from a central service registry is similar for all communication patterns.

This chapter has provided an overview of the fundamental communication patterns of cloud native applications. The next chapter covers how to build connectivity among the microservices of a cloud native application and how to create composition by integrating those services and systems.