Table of Contents

Azure Application Architecture Guide

A guide to designing scalable, resilient, and highly available applications, based on proven practices that we have learned from customer engagements.

Reference Architectures

A set of recommended architectures for Azure. Each architecture includes best practices, prescriptive steps, and a deployable solution.

Cloud Design Patterns

Design patterns for developers and solution architects. Each pattern describes a problem, a pattern that addresses the problem, and an example based on Azure.

Building Microservices on Azure

This multi-part series takes you through the process of designing and building a microservices architecture on Azure. A reference implementation is included.

Azure Data Architecture Guide

A structured approach to designing data-centric solutions on Microsoft Azure.

Best Practices for Cloud Applications

Best practices for cloud applications, covering aspects such as auto-scaling, caching, data partitioning, API design, and others.

Designing for Resiliency

Learn how to design resilient applications for Azure.

Azure Building Blocks

Simplify deployment of Azure resources. With a single settings file, deploy complex architectures in Azure.

Design Review Checklists

Checklists to assist developers and solution architects during the design process.

Azure Virtual Datacenter

When deploying enterprise workloads to the cloud, organizations must balance governance with developer agility. Azure Virtual Datacenter provides models to achieve this balance with an emphasis on governance.

Azure for AWS Professionals

Leverage your AWS experiences in Microsoft Azure.

Performance Antipatterns

How to detect and fix some common causes of performance and scalability problems in cloud applications.

Run SharePoint Server 2016 on Azure

Deploy and run a high availability SharePoint Server 2016 farm on Azure.

Run SAP HANA on Azure

Deploy and run SAP NetWeaver and SAP HANA in a high availability environment on Azure.

Identity Management for Multitenant Applications

Understand the best practices for multitenancy, when using Azure AD for identity management.

Azure Customer Advisory Team

The AzureCAT team's blog

SQL Server Customer Advisory Team

The SQLCAT team's blog

Designing, building, and operating microservices on Azure

1/8/2018 • 6 min to read • Edit Online

Microservices have become a popular architectural style for building cloud applications that are resilient, highly scalable, independently deployable, and able to evolve quickly. To be more than just a buzzword, however, microservices require a different approach to designing and building applications.

In this set of articles, we explore how to build and run a microservices architecture on Azure. Topics include:

- Using Domain Driven Design (DDD) to design a microservices architecture.
- Choosing the right Azure technologies for compute, storage, messaging, and other elements of the design.
- Understanding microservices design patterns.
- Designing for resiliency, scalability, and performance.
- Building a CI/CD pipeline.

Throughout, we focus on an end-to-end scenario: A drone delivery service that lets customers schedule packages to be picked up and delivered via drone. You can find the code for our reference implementation on GitHub

 Reference implementation

But first, let's start with fundamentals. What are microservices, and what are the advantages of adopting a microservices architecture?

Why build microservices?

In a microservices architecture, the application is composed of small, independent services. Here are some of the defining characteristics of microservices:

- Each microservice implements a single business capability.
- A microservice is small enough that a single small team of developers can write and maintain it.
- Microservices run in separate processes, communicating through well-defined APIs or messaging patterns.
- Microservices do not share data stores or data schemas. Each microservice is responsible for managing its own data.
- Microservices have separate code bases, and do not share source code. They may use common utility libraries, however.
- Each microservice can be deployed and updated independently of other services.

Done correctly, microservices can provide a number of useful benefits:

- **Agility.** Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application, and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process; as a result, new features may be held up waiting for a bug fix to be integrated, tested, and published.

- **Small code, small teams.** A microservice should be small enough that a single feature team can build, test, and deploy it. Small code bases are easier to understand. In a large monolithic application, there is a

tendency over time for code dependencies to become tangled, so that adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features. Small team sizes also promote greater agility. The "two-pizza rule" says that a team should be small enough that two pizzas can feed the team. Obviously that's not an exact metric and depends on team appetites! But the point is that large groups tend be less productive, because communication is slower, management overhead goes up, and agility diminishes.

- **Mix of technologies**. Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate.
- **Resiliency**. If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as any upstream microservices are designed to handle faults correctly (for example, by implementing circuit breaking).
- **Scalability**. A microservices architecture allows each microservice to be scaled independently of the others. That lets you scale out subsystems that require more resources, without scaling out the entire application. If you deploy services inside containers, you can also pack a higher density of microservices onto a single host, which allows for more efficient utilization of resources.
- **Data isolation**. It is much easier to perform schema updates, because only a single microservice is impacted. In a monolithic application, schema updates can become very challenging, because different parts of the application may all touch the same data, making any alterations to the schema risky.

No free lunch

These benefits don't come for free. This series of articles is designed to address some of the challenges of building microservices that are resilient, scalable, and manageable.

- **Service boundaries**. When you build microservices, you need to think carefully about where to draw the boundaries between services. Once services are built and deployed in production, it can be hard to refactor across those boundaries. Choosing the right service boundaries is one of the biggest challenges when designing a microservices architecture. How big should each service be? When should functionality be factored across several services, and when should it be kept inside the same service? In this guide, we describe an approach that uses domain-driven design to find service boundaries. It starts with Domain analysis to find the bounded contexts, then applies a set of tactical DDD patterns based on functional and non-functional requirements.
- **Data consistency and integrity**. A basic principle of microservices is that each service manages its own data. This keeps services decoupled, but can lead to challenges with data integrity or redundancy. We explore some of these issues in the Data considerations.
- **Network congestion and latency**. The use of many small, granular services can result in more interservice communication and longer end-to-end latency. The chapter Interservice communication describes considerations for messaging between services. Both synchronous and asynchronous communication have a place in microservices architectures. Good API design is important so that services remain loosely coupled, and can be independently deployed and updated.
- **Complexity**. A microservices application has more moving parts. Each service may be simple, but the services have to work together as a whole. A single user operation may involve multiple services. In the chapter Ingestion and workflow, we examine some of the issues around ingesting requests at high throughput, coordinating a workflow, and handling failures.
- **Communication between clients and the application.** When you decompose an application into many small services, how should clients communicate with those services? Should a client call each individual service directly, or route requests through an API Gateway?
- **Monitoring**. Monitoring a distributed application can be a lot harder than a monolithic application, because

you must correlate telemetry from multiple services. The chapter Logging and monitoring addresses these concerns.

- **Continuous integration and delivery (CI/CD)**. One of the main goals of microservices is agility. To achieve this, you must have automated and robust CI/CD, so that you can quickly and reliably deploy individual services into test and production environments.

The Drone Delivery application

To explore these issues, and to illustrate some of the best practices for a microservices architecture, we created a reference implementation that we call the Drone Delivery application. You can find the reference implementation on GitHub.

Fabrikam, Inc. is starting a drone delivery service. The company manages a fleet of drone aircraft. Businesses register with the service, and users can request a drone to pick up goods for delivery. When a customer schedules a pickup, a backend system assigns a drone and notifies the user with an estimated delivery time. While the delivery is in progress, the customer can track the location of the drone, with a continuously updated ETA.

This scenario involves a fairly complicated domain. Some of the business concerns include scheduling drones, tracking packages, managing user accounts, and storing and analyzing historical data. Moreover, Fabrikam wants to get to market quickly and then iterate quickly, adding new functionality and capabilities. The application needs to operate at cloud scale, with a high service level objective (SLO). Fabrikam also expects that different parts of the system will have very different requirements for data storage and querying. All of these considerations lead Fabrikam to choose a microservices architecture for the Drone Delivery application.

> **NOTE**
>
> For help in choosing between a microservices architecture and other architectural styles, see the Azure Application Architecture Guide.

Our reference implementation uses Kubernetes with Azure Container Service (ACS). However, many of the high-level architectural decisions and challenges will apply to any container orchestrator, including Azure Service Fabric.

Domain analysis

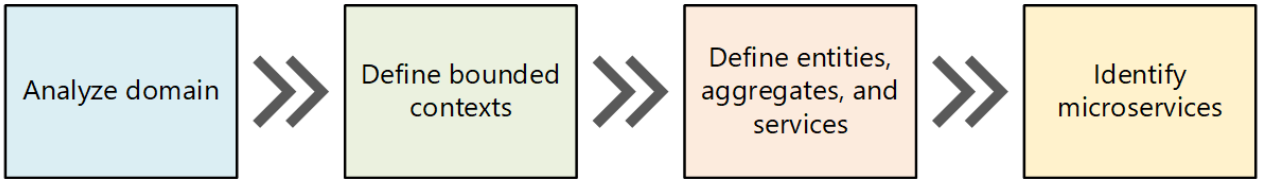
Designing microservices: Domain analysis

1/10/2018 • 13 min to read • Edit Online

One of the biggest challenges of microservices is to define the boundaries of individual services. The general rule is that a service should do "one thing" — but putting that rule into practice requires careful thought. There is no mechanical process that will produce the "right" design. You have to think deeply about your business domain, requirements, and goals. Otherwise, you can end up with a haphazard design that exhibits some undesirable characteristics, such as hidden dependencies between services, tight coupling, or poorly designed interfaces. In this chapter, we take a domain-driven approach to designing microservices.

Microservices should be designed around business capabilities, not horizontal layers such as data access or messaging. In addition, they should have loose coupling and high functional cohesion. Microservices are *loosely coupled* if you can change one service without requiring other services to be updated at the same time. A microservice is *cohesive* if it has a single, well-defined purpose, such as managing user accounts or tracking delivery history. A service should encapsulate domain knowledge and abstract that knowledge from clients. For example, a client should be able to schedule a drone without knowing the details of the scheduling algorithm or how the drone fleet is managed.

Domain-driven design (DDD) provides a framework that can get you most of the way to a set of well-designed microservices. DDD has two distinct phases, strategic and tactical. In strategic DDD, you are defining the large-scale structure of the system. Strategic DDD helps to ensure that your architecture remains focused on business capabilities. Tactical DDD provides a set of design patterns that you can use to create the domain model. These patterns include entities, aggregates, and domain services. These tactical patterns will help you to design microservices that are both loosely coupled and cohesive.



In this chapter and the next, we'll walk through the following steps, applying them to the Drone Delivery application:

1. Start by analyzing the business domain to understand the application's functional requirements. The output of this step is an informal description of the domain, which can be refined into a more formal set of domain models.
2. Next, define the *bounded contexts* of the domain. Each bounded context contains a domain model that represents a particular subdomain of the larger application.
3. Within a bounded context, apply tactical DDD patterns to define entities, aggregates, and domain services.
4. Use the results from the previous step to identify the microservices in your application.

In this chapter, we cover the first three steps, which are primarily concerned with DDD. In the next chapter, we will identify the microservices. However, it's important to remember that DDD is an iterative, ongoing process. Service boundaries aren't fixed in stone. As an application evolves, you may decide to break apart a service into several smaller services.

> **NOTE**
>
> This chapter is not meant to show a complete and comprehensive domain analysis. We deliberately kept the example brief, in order to illustrate the main points. For more background on DDD, we recommend Eric Evans' *Domain-Driven Design*, the book that first introduced the term. Another good reference is *Implementing Domain-Driven Design* by Vaughn Vernon.

Analyze the domain

Using a DDD approach will help you to design microservices so that every service forms a natural fit to a functional business requirement. It can help you to avoid the trap of letting organizational boundaries or technology choices dictate your design.

Before writing any code, you need a bird's eye view of the system that you are creating. DDD starts by modeling the business domain and creating a *domain model*. The domain model is an abstract model of the business domain. It distills and organizes domain knowledge, and provides a common language for developers and domain experts.

Start by mapping all of the business functions and their connections. This will likely be a collaborative effort that involves domain experts, software architects, and other stakeholders. You don't need to use any particular formalism. Sketch a diagram or draw on whiteboard.

As you fill in the diagram, you may start to identify discrete subdomains. Which functions are closely related? Which functions are core to the business, and which provide ancillary services? What is the dependency graph? During this initial phase, you aren't concerned with technologies or implementation details. That said, you should note the place where the application will need to integrate with external systems, such as CRM, payment processing, or billing systems.

Drone Delivery: Analyzing the business domain.

After some initial domain analysis, the Fabrikam team came up with a rough sketch that depicts the Drone Delivery domain.

- **Shipping** is placed in the center of the diagram, because it's core to the business. Everything else in the diagram exists to enable this functionality.
- **Drone management** is also core to the business. Functionality that is closely related to drone management includes **drone repair** and using **predictive analysis** to predict when drones need servicing and maintenance.
- **ETA analysis** provides time estimates for pickup and delivery.
- **Third-party transportation** will enable the application to schedule alternative transportation methods if a package cannot be shipped entirely by drone.
- **Drone sharing** is a possible extension of the core business. The company may have excess drone capacity during certain hours, and could rent out drones that would otherwise be idle. This feature will not be in the initial release.
- **Video surveillance** is another area that the company might expand into later.
- **User accounts**, **Invoicing**, and **Call center** are subdomains that support the core business.

Notice that at this point in the process, we haven't made any decisions about implementation or technologies. Some of the subsystems may involve external software systems or third-party services. Even so, the application needs to interact with these systems and services, so it's important to include them in the domain model.

> **NOTE**
>
> When an application depends on an external system, there is a risk that the external system's data schema or API will leak into your application, ultimately compromising the architectural design. This is particularly true with legacy systems that may not follow modern best practices, and may use convoluted data schemas or obsolete APIs. In that case, it's important to have a well-defined boundary between these external systems and the application. Consider using the Strangler Pattern or the Anti-Corruption Layer Pattern for this purpose.

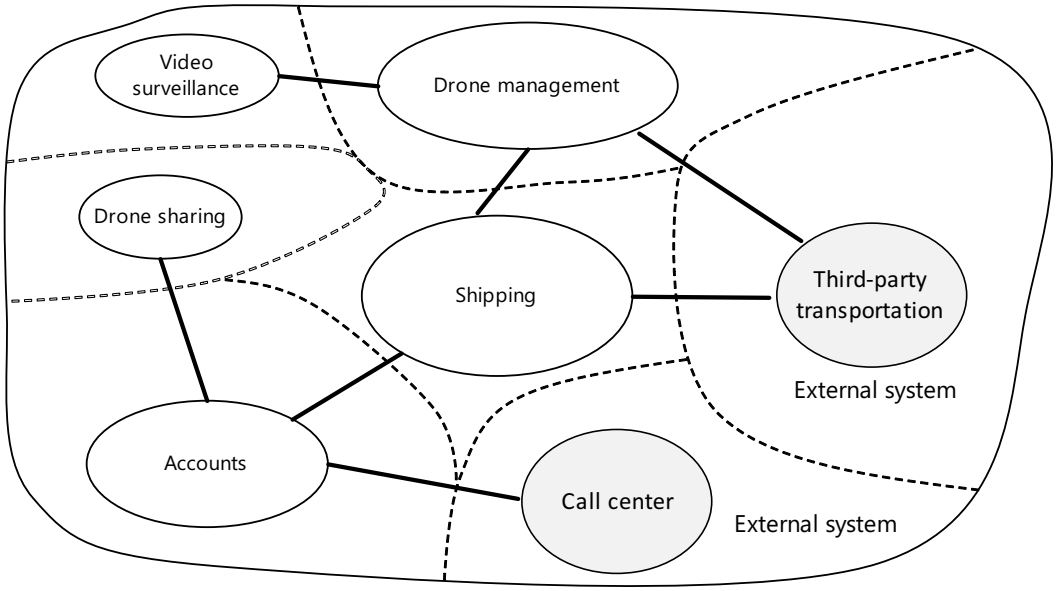
Define bounded contexts

The domain model will include representations of real things in the world — users, drones, packages, and so forth. But that doesn't mean that every part of the system needs to use the same representations for the same things.

For example, subsystems that handle drone repair and predictive analysis will need to represent many physical characteristics drones, such as their maintenance history, mileage, age, model number, performance characteristics, and so on. But when it's time to schedule a delivery, we don't care about those things. The scheduling subsystem only needs to know whether a drone is available, and the ETA for pickup and delivery.

If we tried to create a single model for both of these subsystems, it would be unnecessarily complex. It would also become harder for the model to evolve over time, because any changes will need to satisfy multiple teams working on separate subsystems. Therefore, it's often better to design separate models that represent the same real-world entity (in this case, a drone) in two different contexts. Each model contains only the features and attributes that are relevant within its particular context.

This is where the DDD concept of *bounded contexts* comes into play. A bounded context is simply the boundary within a domain where a particular domain model applies. Looking at the previous diagram, we can group functionality according to whether various functions will share a single domain model.



Bounded contexts are not necessarily isolated from one another. In this diagram, the solid lines connecting the bounded contexts represent places where two bounded contexts interact. For example, Shipping depends on User Accounts to get information about customers, and on Drone Management to schedule drones from the fleet.

In the book *Domain Driven Design*, Eric Evans describes several patterns for maintaining the integrity of a domain model when it interacts with another bounded context. One of the main principles of microservices is that services communicate through well-defined APIs. This approach corresponds to two patterns that Evans calls Open Host Service and Published Language. The idea of Open Host Service is that a subsystem defines a formal protocol (API) for other subsystems to communicate with it. Published Language extends this idea by publishing the API in a form that other teams can use to write clients. In the chapter on API Design, we discuss using OpenAPI

Specification (formerly known as Swagger) to define language-agnostic interface descriptions for REST APIs, expressed in JSON or YAML format.

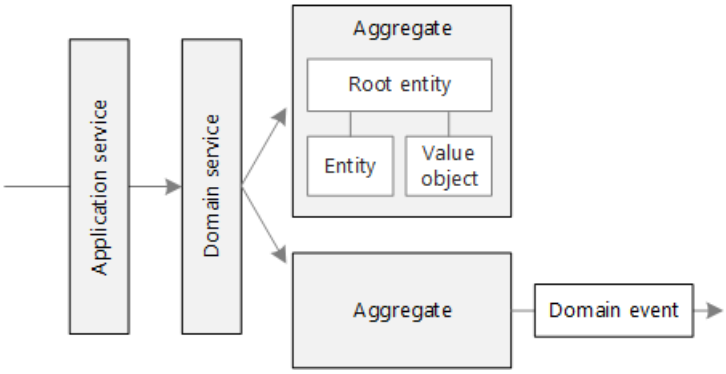
For the rest of this journey, we will focus on the Shipping bounded context.

Tactical DDD

During the strategic phase of DDD, you are mapping out the business domain and defining bounded contexts for your domain models. Tactical DDD is when you define your domain models with more precision. The tactical patterns are applied within a single bounded context. In a microservices architecture, we are particularly interested in the entity and aggregate patterns. Applying these patterns will help us to identify natural boundaries for the services in our application (see next chapter). As a general principle, a microservice should be no smaller than an aggregate, and no larger than a bounded context. First, we'll review the tactical patterns. Then we'll apply them to the Shipping bounded context in the Drone Delivery application.

Overview of the tactical patterns

This section provides a brief summary of the tactical DDD patterns, so if you are already familiar with DDD, you can probably skip this section. The patterns are described in more detail in chapters 5 – 6 of Eric Evans' book, and in *Implementing Domain-Driven Design* by Vaughn Vernon.



Entities. An entity is an object with a unique identity that persists over time. For example, in a banking application, customers and accounts would be entities.

- An entity has a unique identifier in the system, which can be used to look up or retrieve the entity. That doesn't mean the identifier is always exposed directly to users. It could be a GUID or a primary key in a database.
- An identity may span multiple bounded contexts, and may endure beyond the lifetime of the application. For example, bank account numbers or government-issued IDs are not tied to the lifetime of a particular application.
- The attributes of an entity may change over time. For example, a person's name or address might change, but they are still the same person.
- An entity can hold references to other entities.

Value objects. A value object has no identity. It is defined only by the values of its attributes. Value objects are also immutable. To update a value object, you always create a new instance to replace the old one. Value objects can have methods that encapsulate domain logic, but those methods should have no side-effects on the object's state. Typical examples of value objects include colors, dates and times, and currency values.

Aggregates. An aggregate defines a consistency boundary around one or more entities. Exactly one entity in an aggregate is the root. Lookup is done using the root entity's identifier. Any other entities in the aggregate are children of the root, and are referenced by following pointers from the root.

The purpose of an aggregate is to model transactional invariants. Things in the real world have complex webs of relationships. Customers create orders, orders contain products, products have suppliers, and so on. If the application modifies several related objects, how does it guarantee consistency? How do we keep track of invariants and enforce them?

Traditional applications have often used database transactions to enforce consistency. In a distributed application, however, that's often not feasible. A single business transaction may span multiple data stores, or may be long running, or may involve third-party services. Ultimately it's up to the application, not the data layer, to enforce the invariants required for the domain. That's what aggregates are meant to model.

> **NOTE**
>
> An aggregate might consist of a single entity, without child entities. What makes it an aggregate is the transactional boundary.

Domain and application services. In DDD terminology, a service is an object that implements some logic without holding any state. Evans distinguishes between *domain services*, which encapsulate domain logic, and *application services*, which provide technical functionality, such as user authentication or sending an SMS message. Domain services are often used to model behavior that spans multiple entities.

> **NOTE**
>
> The term *service* is overloaded in software development. The definition here is not directly related to microservices.

Domain events. Domain events can be used to notify other parts of the system when something happens. As the name suggests, domain events should mean something within the domain. For example, "a record was inserted into a table" is not a domain event. "A delivery was cancelled" is a domain event. Domain events are especially relevant in a microservices architecture. Because microservices are distributed and don't share data stores, domain events provide a way for microservices to coordinate with each other. The chapter Interservice communication discusses asynchronous messaging in more detail.

There are a few other DDD patterns not listed here, including factories, repositories, and modules. These can be useful patterns for when you are implementing a microservice, but they are less relevant when designing the boundaries between microservice.

Drone delivery: Applying the patterns

We start with the scenarios that the Shipping bounded context must handle.

- A customer can request a drone to pick up goods from a business that is registered with the drone delivery service.
- The sender generates a tag (barcode or RFID) to put on the package.
- A drone will pick up and deliver a package from the source location to the destination location.
- When a customer schedules a delivery, the system provides an ETA based on route information, weather conditions, and historical data.
- When the drone is in flight, a user can track the current location and the latest ETA.
- Until a drone has picked up the package, the customer can cancel a delivery.
- The customer is notified when the delivery is completed.
- The sender can request delivery confirmation from the customer, in the form of a signature or finger print.
- Users can look up the history of a completed delivery.

From these scenarios, the development team identified the following **entities**.

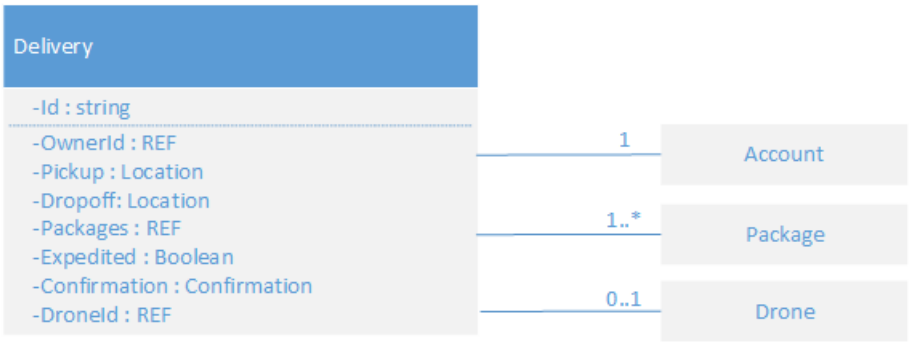
- Delivery
- Package
- Drone
- Account

- Confirmation
- Notification
- Tag

The first four, Delivery, Package, Drone, and Account, are all **aggregates** that represent transactional consistency boundaries. Confirmations and Notifications are child entities of Deliveries, and Tags are child entities of Packages.

The **value objects** in this design include Location, ETA, PackageWeight, and PackageSize.

To illustrate, here is a UML diagram of the Delivery aggregate. Notice that it holds references to other aggregates, including Account, Package, and Drone.

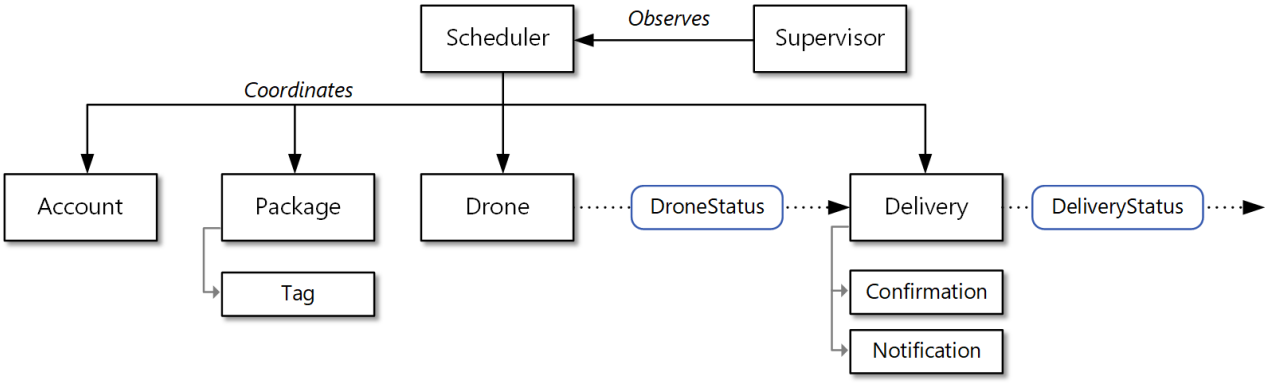


There are two domain events:

- While a drone is in flight, the Drone entity sends DroneStatus events that describe the drone's location and status (in-flight, landed).
- The Delivery entity sends DeliveryTracking events whenever the stage of a delivery changes. These include DeliveryCreated, DeliveryRescheduled, DeliveryHeadedToDropoff, and DeliveryCompleted.

Notice that these events describe things that are meaningful within the domain model. They describe something about the domain, and aren't tied to a particular programming language construct.

The development team identified one more area of functionality, which doesn't fit neatly into any of the entities described so far. Some part of the system must coordinate all of the steps involved in scheduling or updating a delivery. Therefore, the development team added two **domain services** to the design: a *Scheduler* that coordinates the steps, and a *Supervisor* that monitors the status of each step, in order to detect whether any steps have failed or timed out. This is a variation of the Scheduler Agent Supervisor pattern.

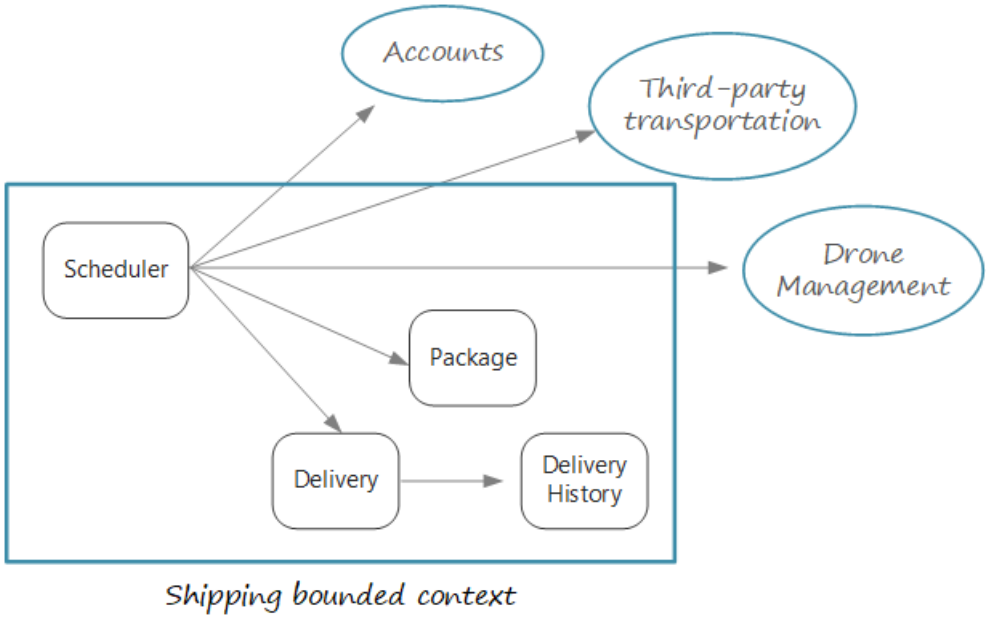


Identifying microservice boundaries

Designing microservices: Identifying microservice boundaries

5/4/2018 • 9 min to read • Edit Online

What is the right size for a microservice? You often hear something to the effect of, "not too big and not too small" — and while that's certainly correct, it's not very helpful in practice. But if you start from a carefully designed domain model, it's much easier to reason about microservices.



From domain model to microservices

In the previous chapter, we defined a set of bounded contexts for the Drone Delivery application. Then we looked more closely at one of these bounded contexts, the Shipping bounded context, and identified a set of entities, aggregates, and domain services for that bounded context.

Now we're ready to go from domain model to application design. Here's an approach that you can use to derive microservices from the domain model.

1. Start with a bounded context. In general, the functionality in a microservice should not span more than one bounded context. By definition, a bounded context marks the boundary of a particular domain model. If you find that a microservice mixes different domain models together, that's a sign that you may need to go back and refine your domain analysis.

2. Next, look at the aggregates in your domain model. Aggregates are often good candidates for microservices. A well-designed aggregate exhibits many of the characteristics of a well-designed microservice, such as:

   - An aggregate is derived from business requirements, rather than technical concerns such as data access or messaging.
   - An aggregate should have high functional cohesion.
   - An aggregate is a boundary of persistence.
   - Aggregates should be loosely coupled.

3. Domain services are also good candidates for microservices. Domain services are stateless operations across multiple aggregates. A typical example is a workflow that involves several microservices. We'll see

an example of this in the Drone Delivery application.

4. Finally, consider non-functional requirements. Look at factors such as team size, data types, technologies, scalability requirements, availability requirements, and security requirements. These factors may lead you to further decompose a microservice into two or more smaller services, or do the opposite and combine several microservices into one.

After you identify the microservices in your application, validate your design against the following criteria:

- Each service has a single responsibility.
- There are no chatty calls between services. If splitting functionality into two services causes them to be overly chatty, it may be a symptom that these functions belong in the same service.
- Each service is small enough that it can be built by a small team working independently.
- There are no inter-dependencies that will require two or more services to be deployed in lock-step. It should always be possible to deploy a service without redeploying any other services.
- Services are not tightly coupled, and can evolve independently.
- Your service boundaries will not create problems with data consistency or integrity. Sometimes it's important to maintain data consistency by putting functionality into a single microservice. That said, consider whether you really need strong consistency. There are strategies for addressing eventual consistency in a distributed system, and the benefits of decomposing services often outweigh the challenges of managing eventual consistency.

Above all, it's important to be pragmatic, and remember that domain-driven design is an iterative process. When in doubt, start with more coarse-grained microservices. Splitting a microservice into two smaller services is easier than refactoring functionality across several existing microservices.

Drone Delivery: Defining the microservices

Recall that the development team had identified the four aggregates — Delivery, Package, Drone, and Account — and two domain services, Scheduler and Supervisor.

Delivery and Package are obvious candidates for microservices. The Scheduler and Supervisor coordinate the activities performed by other microservices, so it makes sense to implement these domain services as microservices.

Drone and Account are interesting because they belong to other bounded contexts. One option is for the Scheduler to call the Drone and Account bounded contexts directly. Another option is to create Drone and Account microservices inside the Shipping bounded context. These microservices would mediate between the bounded contexts, by exposing APIs or data schemas that are more suited to the Shipping context.

The details of the Drone and Account bounded contexts are beyond the scope of this guidance, so we created mock services for them in our reference implementation. But here are some factors to consider in this situation:

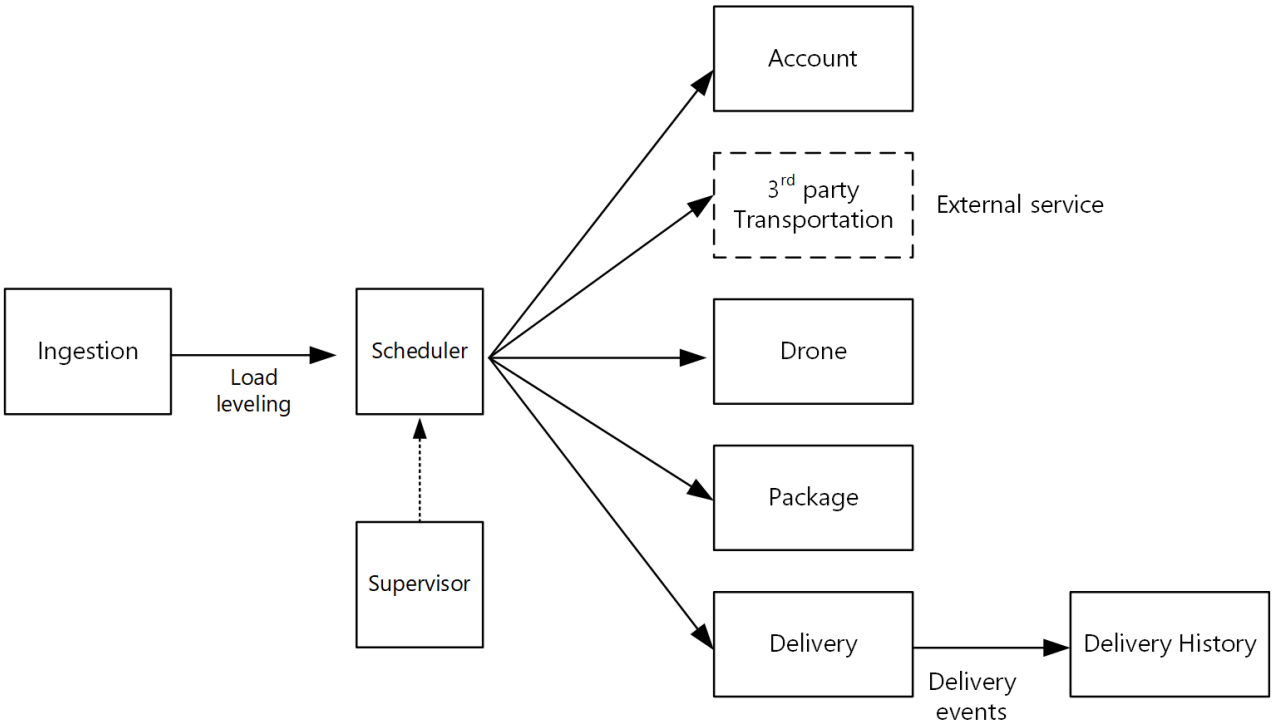
- What is the network overhead of calling directly into the other bounded context?
- Is the data schema for the other bounded context suitable for this context, or is it better to have a schema that's tailored to this bounded context?
- Is the other bounded context a legacy system? If so, you might create a service that acts as an anti-corruption layer to translate between the legacy system and the modern application.
- What is the team structure? Is it easy to communicate with the team that's responsible for the other bounded context? If not, creating a service that mediates between the two contexts can help to mitigate the cost of cross-team communication.

So far, we haven't considered any non-functional requirements. Thinking about the application's throughput requirements, the development team decided to create a separate Ingestion microservice that is responsible for

ingesting client requests. This microservice will implement load leveling by putting incoming requests into a buffer for processing. The Scheduler will read the requests from the buffer and execute the workflow.

Non-functional requirements led the team to create one additional service. All of the services so far have been about the process of scheduling and delivering packages in real time. But the system also needs to store the history of every delivery in long-term storage for data analysis. The team considered making this the responsibility of the Delivery service. However, the data storage requirements are quite different for historical analysis versus in-flight operations (see Data considerations). Therefore, the team decided to create a separate Delivery History service, which will listen for DeliveryTracking events from the Delivery service and write the events into long-term storage.

The following diagram shows the design at this point:



Choosing a compute option

The term *compute* refers to the hosting model for the computing resources that your application runs on. For a microservices architecture, two approaches are especially popular:

- A service orchestrator that manages services running on dedicated nodes (VMs).
- A serverless architecture using functions as a service (FaaS).

While these aren't the only options, they are both proven approaches to building microservices. An application might include both approaches.

Service orchestrators

An orchestrator handles tasks related to deploying and managing a set of services. These tasks include placing services on nodes, monitoring the health of services, restarting unhealthy services, load balancing network traffic across service instances, service discovery, scaling the number of instances of a service, and applying configuration updates. Popular orchestrators include Kubernetes, DC/OS, Docker Swarm, and Service Fabric.

- Azure Container Service (ACS) is an Azure service that lets you deploy a production-ready Kubernetes, DC/OS, or Docker Swarm cluster.
- AKS (Azure Container Service) is a managed Kubernetes service. AKS provisions Kubernetes and exposes the Kubernetes API endpoints, but hosts and manages the Kubernetes control plane, performing automated upgrades, automated patching, autoscaling, and other management tasks. You can think of AKS

as being "Kubernetes APIs as a service." At the time of writing, AKS is still in preview. However, it's expected that AKS will become the preferred way to run Kubernetes in Azure.

- Service Fabric is a distributed systems platform for packaging, deploying, and managing microservices. Microservices can be deployed to Service Fabric as containers, as binary executables, or as Reliable Services. Using the Reliable Services programming model, services can directly use Service Fabric programming APIs to query the system, report health, receive notifications about configuration and code changes, and discover other services. A key differentiation with Service Fabric is its strong focus on building stateful services using Reliable Collections.

Containers

Sometimes people talk about containers and microservices as if they were the same thing. While that's not true — you don't need containers to build microservices — containers do have some benefits that are particularly relevant to microservices, such as:

- **Portability**. A container image is a standalone package that runs without needing to install libraries or other dependencies. That makes them easy to deploy. Containers can be started and stopped quickly, so you can spin up new instances to handle more load or to recover from node failures.
- **Density**. Containers are lightweight compared with running a virtual machine, because they share OS resources. That makes it possible to pack multiple containers onto a single node, which is especially useful when the application consists of many small services.
- **Resource isolation**. You can limit the amount of memory and CPU that is available to a container, which can help to ensure that a runaway process doesn't exhaust the host resources. See the Bulkhead Pattern for more information.

Serverless (Functions as a Service)

With a serverless architecture, you don't manage the VMs or the virtual network infrastructure. Instead, you deploy code and the hosting service handles putting that code onto a VM and executing it. This approach tends to favor small granular functions that are coordinated using event-based triggers. For example, a message being placed onto a queue might trigger a function that reads from the queue and processes the message.

Azure Functions is a serverless compute service that supports various function triggers, including HTTP requests, Service Bus queues, and Event Hubs events. For a complete list, see Azure Functions triggers and bindings concepts. Also consider Azure Event Grid, which is a managed event routing service in Azure.

Orchestrator or serverless?

Here are some factors to consider when choosing between an orchestrator approach and a serverless approach.

Manageability A serverless application is easy to manage, because the platform manages all the of compute resources for you. While an orchestrator abstracts some aspects of managing and configuring a cluster, it does not completely hide the underlying VMs. With an orchestrator, you will need to think about issues such as load balancing, CPU and memory usage, and networking.

Flexibility and control. An orchestrator gives you a great deal of control over configuring and managing your services and the cluster. The tradeoff is additional complexity. With a serverless architecture, you give up some degree of control because these details are abstracted.

Portability. All of the orchestrators listed here (Kubernetes, DC/OS, Docker Swarm, and Service Fabric) can run on-premises or in multiple public clouds.

Application integration. It can be challenging to build a complex application using a serverless architecture. One option in Azure is to use Azure Logic Apps to coordinate a set of Azure Functions. For an example of this approach, see Create a function that integrates with Azure Logic Apps.

Cost. With an orchestrator, you pay for the VMs that are running in the cluster. With a serverless application, you

pay only for the actual compute resources consumed. In both cases, you need to factor in the cost of any additional services, such as storage, databases, and messaging services.

Scalability. Azure Functions scales automatically to meet demand, based on the number of incoming events. With an orchestrator, you can scale out by increasing the number of service instances running in the cluster. You can also scale by adding additional VMs to the cluster.

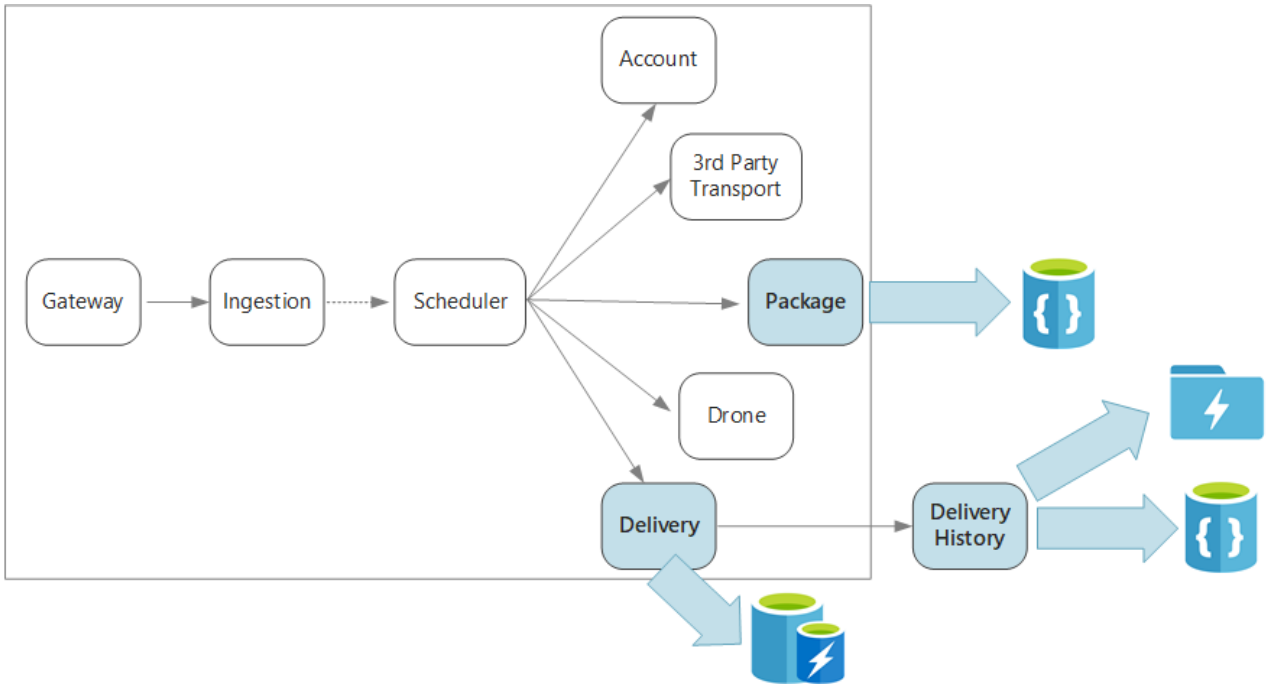
Our reference implementation primarily uses Kubernetes, but we did use Azure Functions for one service, namely the Delivery History service. Azure Functions was a good fit for this particular service, because it's is an event-driven workload. By using an Event Hubs trigger to invoke the function, the service needed a minimal amount of code. Also, the Delivery History service is not part of the main workflow, so running it outside of the Kubernetes cluster doesn't affect the end-to-end latency of user-initiated operations.

Data considerations

Designing microservices: Data considerations

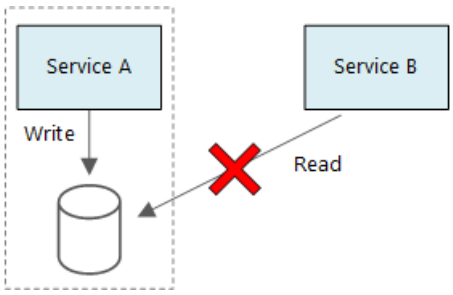
12/9/2017 • 7 min to read • Edit Online

This chapter describes considerations for managing data in a microservices architecture. Because every microservice manages its own data, data integrity and data consistency are critical challenges.



A basic principle of microservices is that each service manages its own data. Two services should not share a data store. Instead, each service is responsible for its own private data store, which other services cannot access directly.

The reason for this rule is to avoid unintentional coupling between services, which can result if services share the same underlying data schemas. If there is a change to the data schema, the change must be coordinated across every service that relies on that database. By isolating each service's data store, we can limit the scope of change, and preserve the agility of truly independent deployments. Another reason is that each microservice may have its own data models, queries, or read/write patterns. Using a shared data store limits each team's ability to optimize data storage for their particular service.



This approach naturally leads to polyglot persistence — the use of multiple data storage technologies within a single application. One service might require the schema-on-read capabilities of a document database. Another might need the referential integrity provided by an RDBMS. Each team is free to make the best choice for their service. For more about the general principle of polyglot persistence, see Use the best data store for the job.

> **NOTE**
>
> It's fine for services to share the same physical database server. The problem occurs when services share the same schema, or read and write to the same set of database tables.

Challenges

Some challenges arise from this distributed approach to managing data. First, there may be redundancy across the data stores, with the same item of data appearing in multiple places. For example, data might be stored as part of a transaction, then stored elsewhere for analytics, reporting, or archiving. Duplicated or partitioned data can lead to issues of data integrity and consistency. When data relationships span multiple services, you can't use traditional data management techniques to enforce the relationships.

Traditional data modeling uses the rule of "one fact in one place." Every entity appears exactly once in the schema. Other entities may hold references to it but not duplicate it. The obvious advantage to the traditional approach is that updates are made in a single place, which avoids problems with data consistency. In a microservices architecture, you have to consider how updates are propagated across services, and how to manage eventual consistency when data appears in multiple places without strong consistency.

Approaches to managing data

There is no single approach that's correct in all cases, but here are some general guidelines for managing data in a microservices architecture.

- Embrace eventual consistency where possible. Understand the places in the system where you need strong consistency or ACID transactions, and the places where eventual consistency is acceptable.
- When you need strong consistency guarantees, one service may represent the source of truth for a given entity, which is exposed through an API. Other services might hold their own copy of the data, or a subset of the data, that is eventually consistent with the master data but not considered the source of truth. For example, imagine an e-commerce system with a customer order service and a recommendation service. The recommendation service might listen to events from the order service, but if a customer requests a refund, it is the order service, not the recommendation service, that has the complete transaction history.
- For transactions, use patterns such as Scheduler Agent Supervisor and Compensating Transaction to keep data consistent across several services. You may need to store an additional piece of data that captures the state of a unit of work that spans multiple services, to avoid partial failure among multiple services. For example, keep a work item on a durable queue while a multi-step transaction is in progress.
- Store only the data that a service needs. A service might only need a subset of information about a domain entity. For example, in the Shipping bounded context, we need to know which customer is associated to a particular delivery. But we don't need the customer's billing address — that's managed by the Accounts bounded context. Thinking carefully about the domain, and using a DDD approach, can help here.
- Consider whether your services are coherent and loosely coupled. If two services are continually exchanging information with each other, resulting in chatty APIs, you may need to redraw your service boundaries, by merging two services or refactoring their functionality.
- Use an event driven architecture style. In this architecture style, a service publishes an event when there are changes to its public models or entities. Interested services can subscribe to these events. For example, another service could use the events to construct a materialized view of the data that is more suitable for querying.
- A service that owns events should publish a schema that can be used to automate serializing and deserializing the events, to avoid tight coupling between publishers and subscribers. Consider JSON

schema or a framework like Microsoft Bond, Protobuf, or Avro.

- At high scale, events can become a bottleneck on the system, so consider using aggregation or batching to reduce the total load.

Drone Delivery: Choosing the data stores

Even with only a few services, the Shipping bounded context illustrates several of the points discussed in this section.

When a user schedules a new delivery, the client request includes information about the both the delivery, such as the pickup and dropoff locations, and about the package, such as the size and weight. This information defines a unit of work, which the Ingestion service sends to Event Hubs. It's important that the unit of work stays in persistent storage while the Scheduler service is executing the workflow, so that no delivery requests are lost. For more discussion of the workflow, see Ingestion and workflow.

The various backend services care about different portions of the information in the request, and also have different read and write profiles.

Delivery service

The Delivery service stores information about every delivery that is currently scheduled or in progress. It listens for events from the drones, and tracks the status of deliveries that are in progress. It also sends domain events with delivery status updates.

It's expected that users will frequently check the status of a delivery while they are waiting for their package. Therefore, the Delivery service requires a data store that emphasizes throughput (read and write) over long-term storage. Also, the Delivery service does not perform any complex queries or analysis, it simply fetches the latest status for a given delivery. The Delivery service team chose Azure Redis Cache for its high read-write performance. The information stored in Redis is relatively short-lived. Once a delivery is complete, the Delivery History service is the system of record.

Delivery History service

The Delivery History service listens for delivery status events from the Delivery service. It stores this data in long-term storage. There are two different use-cases for this historical data, which have different data storage requirements.

The first scenario is aggregating the data for the purpose of data analytics, in order to optimize the business or improve the quality of the service. Note that the Delivery History service doesn't perform the actual analysis of the data. It's only responsible for the ingestion and storage. For this scenario, the storage must be optimized for data analysis over a large set of data, using a schema-on-read approach to accommodate a variety of data sources. Azure Data Lake Store is a good fit for this scenario. Data Lake Store is an Apache Hadoop file system compatible with Hadoop Distributed File System (HDFS), and is tuned for performance for data analytics scenarios.

The other scenario is enabling users to look up the history of a delivery after the delivery is completed. Azure Data Lake is not particularly optimized for this scenario. For optimal performance, Microsoft recommends storing time-series data in Data Lake in folders partitioned by date. (See Tuning Azure Data Lake Store for performance). However, that structure is not optimal for looking up individual records by ID. Unless you also know the timestamp, a lookup by ID requires scanning the entire collection. Therefore, the Delivery History service also stores a subset of the historical data in Cosmos DB for quicker lookup. The records don't need to stay in Cosmos DB indefinitely. Older deliveries can be archived — say, after a month. This could be done by running an occasional batch process.

Package service

The Package service stores information about all of the packages. The storage requirements for the Package are:

- Long-term storage.

- Able to handle a high volume of packages, requiring high write throughput.
- Support simple queries by package ID. No complex joins or requirements for referential integrity.

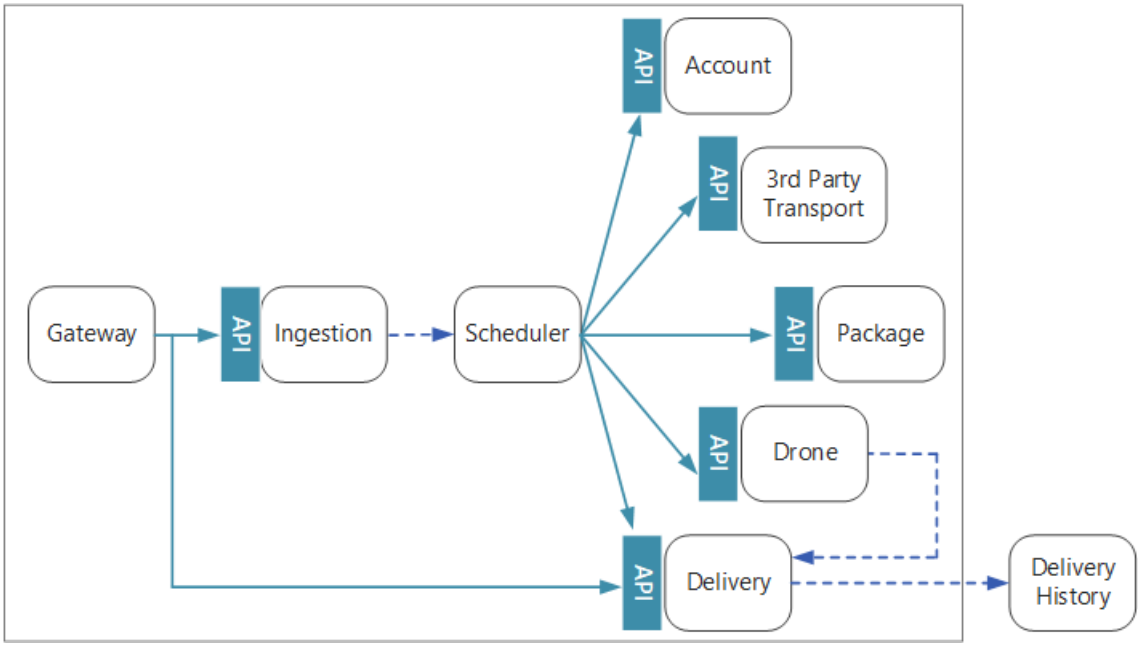
Because the package data is not relational, a document oriented database is appropriate, and Cosmos DB can achieve very high throughput by using sharded collections. The team that works on the Package service is familiar with the MEAN stack (MongoDB, Express.js, AngularJS, and Node.js), so they select the MongoDB API for Cosmos DB. That lets them leverage their existing experience with MongoDB, while getting the benefits of Cosmos DB, which is a managed Azure service.

Interservice communication

Designing microservices: Interservice communication

12/29/2017 • 10 min to read • Edit Online

Communication between microservices must be efficient and robust. With lots of small services interacting to complete a single transaction, this can be a challenge. In this chapter, we look at the tradeoffs between asynchronous messaging versus synchronous APIs. Then we look at some of the challenges in designing resilient interservice communication, and the role that a service mesh can play.



Challenges

Here are some of the main challenges arising from service-to-service communication. Service meshes, described later in this chapter, are designed to handle many of these challenges.

Resiliency. There may be dozens or even hundreds of instances of any given microservice. An instance can fail for any number of reasons. There can be a node-level failure, such as a hardware failure or a VM reboot. An instance might crash, or be overwhelmed with requests and unable to process any new requests. Any of these events can cause a network call to fail. There are two design patterns that can help make service-to-service network calls more resilient:

- **Retry**. A network call may fail because of a transient fault that goes away by itself. Rather than fail outright, the caller should typically retry the operation a certain number of times, or until a configured time-out period elapses. However, if an operation is not idempotent, retries can cause unintended side effects. The original call might succeed, but the caller never gets a response. If the caller retries, the operation may be invoked twice. Generally, it's not safe to retry POST or PATCH methods, because these are not guaranteed to be idempotent.
- **Circuit Breaker**. Too many failed requests can cause a bottleneck, as pending requests accumulate in the queue. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on, which can cause cascading failures. The Circuit Breaker pattern can prevent a service from repeatedly trying an operation that is likely to fail.

Load balancing. When service "A" calls service "B", the request must reach a running instance of service "B". In Kubernetes, the `Service` resource type provides a stable IP address for a group of pods. Network traffic to the

service's IP address gets forwarded to a pod by means of iptable rules. By default, a random pod is chosen. A service mesh (see below) can provide more intelligent load balancing algorithms based on observed latency or other metrics.

Distributed tracing. A single transaction may span multiple services. That can make it hard to monitor the overall performance and health of the system. Even if every service generates logs and metrics, without some way to tie them together, they are of limited use. The chapter Logging and monitoring talks more about distributed tracing, but we mention it here as a challenge.

Service versioning. When a team deploys a new version of a service, they must avoid breaking any other services or external clients that depend on it. In addition, you might want to run multiple versions of a service side-by-side, and route requests to a particular version. See API Versioning for more discussion of this issue.

TLS encryption and mutual TLS authentication. For security reasons, you may want to encrypt traffic between services with TLS, and use mutual TLS authentication to authenticate callers.

Synchronous versus asynchronous messaging

There are two basic messaging patterns that microservices can use to communicate with other microservices.

1. Synchronous communication. In this pattern, a service calls an API that another service exposes, using a protocol such as HTTP or gRPC. This option is a synchronous messaging pattern because the caller waits for a response from the receiver.
2. Asynchronous message passing. In this pattern, a service sends message without waiting for a response, and one or more services process the message asynchronously.

It's important to distinguish between asynchronous I/O and an asynchronous protocol. Asynchronous I/O means the calling thread is not blocked while the I/O completes. That's important for performance, but is an implementation detail in terms of the architecture. An asynchronous protocol means the sender doesn't wait for a response. HTTP is a synchronous protocol, even though an HTTP client may use asynchronous I/O when it sends a request.

There are tradeoffs to each pattern. Request/response is a well-understood paradigm, so designing an API may feel more natural than designing a messaging system. However, asynchronous messaging has some advantages that can be very useful in a microservices architecture:

- **Reduced coupling**. The message sender does not need to know about the consumer.
- **Multiple subscribers**. Using a pub/sub model, multiple consumers can subscribe to receive events. See Event-driven architecture style.
- **Failure isolation**. If the consumer fails, the sender can still send messages. The messages will be picked up when the consumer recovers. This ability is especially useful in a microservices architecture, because each service has its own lifecycle. A service could become unavailable or be replaced with a newer version at any given time. Asynchronous messaging can handle intermittent downtime. Synchronous APIs, on the other hand, require the downstream service to be available or the operation fails.
- **Responsiveness**. An upstream service can reply faster if it does not wait on downstream services. This is especially useful in a microservices architecture. If there is a chain of service dependencies (service A calls B, which calls C, and so on), waiting on synchronous calls can add unacceptable amounts of latency.
- **Load leveling**. A queue can act as a buffer to level the workload, so that receivers can process messages at their own rate.
- **Workflows**. Queues can be used to manage a workflow, by check-pointing the message after each step in the workflow.

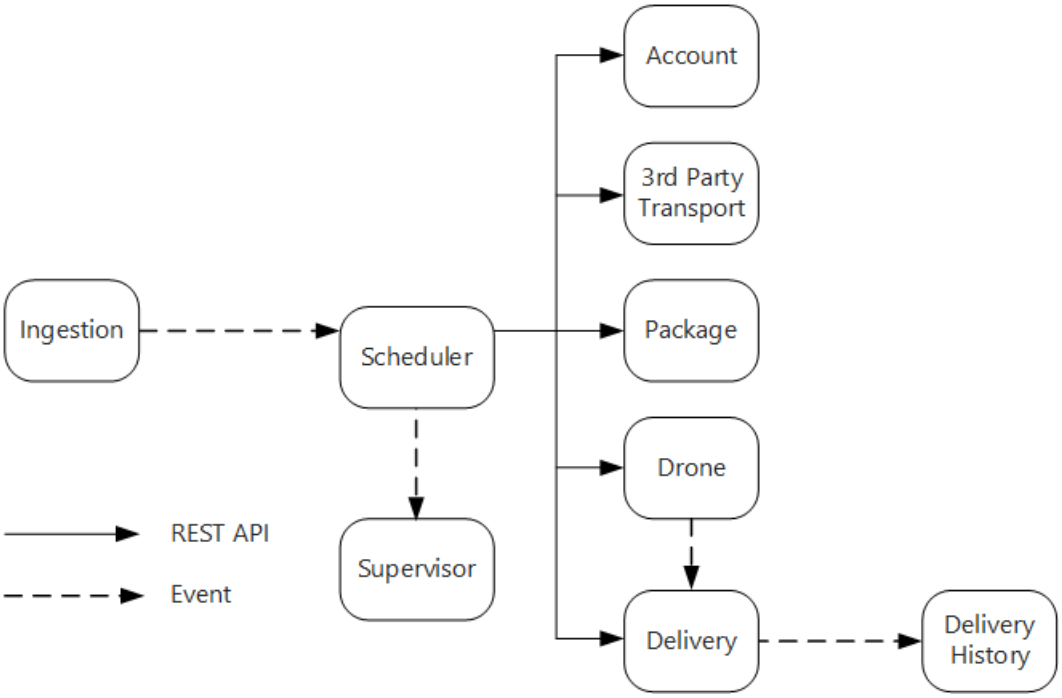
However, there are also some challenges to using asynchronous messaging effectively.

- **Coupling with the messaging infrastructure**. Using a particular messaging infrastructure may cause tight coupling with that infrastructure. It will be difficult to switch to another messaging infrastructure later.
- **Latency**. End-to-end latency for an operation may become high if the message queues fill up.
- **Cost**. At high throughputs, the monetary cost of the messaging infrastructure could be significant.
- **Complexity**. Handling asynchronous messaging is not a trivial task. For example, you must handle duplicated messages, either by de-duplicating or by making operations idempotent. It's also hard to implement request-response semantics using asynchronous messaging. To send a response, you need another queue, plus a way to correlate request and response messages.
- **Throughput**. If messages require *queue semantics*, the queue can become a bottleneck in the system. Each message requires at least one queue operation and one dequeue operation. Moreover, queue semantics generally require some kind of locking inside the messaging infrastructure. If the queue is a managed service, there may be additional latency, because the queue is external to the cluster's virtual network. You can mitigate these issues by batching messages, but that complicates the code. If the messages don't require queue semantics, you might be able to use an event *stream* instead of a queue. For more information, see Event-driven architectural style.

Drone Delivery: Choosing the messaging patterns

With these considerations in mind, the development team made the following design choices for the Drone Delivery application

- The Ingestion service exposes a public REST API that client applications use to schedule, update, or cancel deliveries.
- The Ingestion service uses Event Hubs to send asynchronous messages to the Scheduler service. Asynchronous messages are necessary to implement the load-leveling that is required for ingestion. For details on how the Ingestion and Scheduler services interact, see Ingestion and workflow.
- The Account, Delivery, Package, Drone, and Third-party Transport services all expose internal REST APIs. The Scheduler service calls these APIs to carry out a user request. One reason to use synchronous APIs is that the Scheduler needs to get a response from each of the downstream services. A failure in any of the downstream services means the entire operation failed. However, a potential issue is the amount of latency that is introduced by calling the backend services.
- If any downstream service has a non-transient failure, the entire transaction should be marked as failed. To handle this case, the Scheduler service sends an asynchronous message to the Supervisor, so that the Supervisor can schedule compensating transactions, as described in the chapter Ingestion and workflow.
- The Delivery service exposes a public API that clients can use to get the status of a delivery. In the chapter API gateway, we discuss how an API gateway can hide the underlying services from the client, so the client doesn't need to know which services expose which APIs.
- While a drone is in flight, the Drone service sends events that contain the drone's current location and status. The Delivery service listens to these events in order to track the status of a delivery.
- When the status of a delivery changes, the Delivery service sends a delivery status event, such as `DeliveryCreated` or `DeliveryCompleted`. Any service can subscribe to these events. In the current design, the Delivery service is the only subscriber, but there might be other subscribers later. For example, the events might go to a real-time analytics service. And because the Scheduler doesn't have to wait for a response, adding more subscribers doesn't affect the main workflow path.



Notice that delivery status events are derived from drone location events. For example, when a drone reaches a delivery location and drops off a package, the Delivery service translates this into a DeliveryCompleted event. This is an example of thinking in terms of domain models. As described earlier, Drone Management belongs in a separate bounded context. The drone events convey the physical location of a drone. The delivery events, on the other hand, represent changes in the status of a delivery, which is a different business entity.

Using a service mesh

A *service mesh* is a software layer that handles service-to-service communication. Service meshes are designed to address many of the concerns listed in the previous section, and to move responsibility for these concerns away from the microservices themselves and into a shared layer. The service mesh acts as a proxy that intercepts network communication between microservices in the cluster.

> **NOTE**
> Service mesh is an example of the Ambassador pattern — a helper service that sends network requests on behalf of the application.

Right now, the main options for a service mesh in Kubernetes are linkerd and Istio. Both of these technologies are evolving rapidly. At the time we wrote this guide, the latest Istio release is 0.2, so it is still very new. However, some features that both linkerd and Istio have in common include:

- Load balancing at the session level, based on observed latencies or number of outstanding requests. This can improve performance over the layer-4 load balancing that is provided by Kubernetes.
- Layer-7 routing based on URL path, Host header, API version, or other application-level rules.
- Retry of failed requests. A service mesh understands HTTP error codes, and can automatically retry failed requests. You can configure that maximum number of retries, along with a timeout period in order to bound the maximum latency.
- Circuit breaking. If an instance consistently fails requests, the service mesh will temporarily mark it as unavailable. After a backoff period, it will try the instance again. You can configure the circuit breaker based on various criteria, such as the number of consecutive failures,
- Service mesh captures metrics about interservice calls, such as the request volume, latency, error and

success rates, and response sizes. The service mesh also enables distributed tracing by adding correlation information for each hop in a request.

- Mutual TLS Authentication for service-to-service calls.

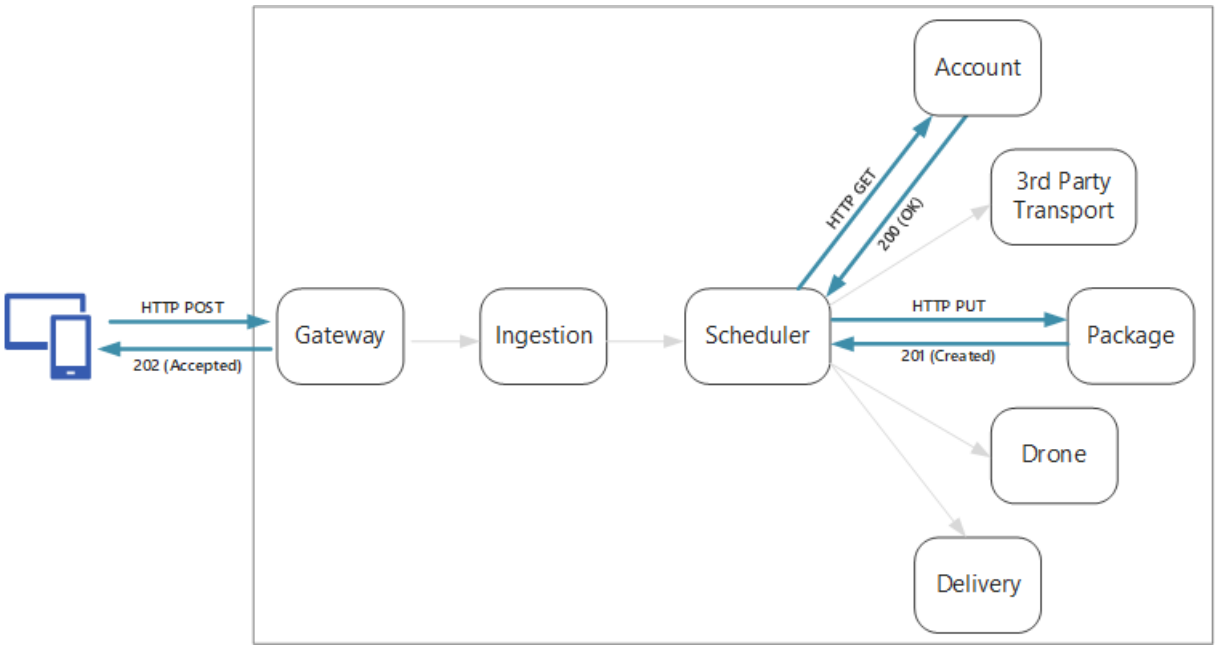
Do you need a service mesh? The value they add to a distributed system is certainly compelling. If you don't have a service mesh, you will need to consider each of the challenges mentioned at the beginning of the chapter. You can solve problems like retry, circuit breaker, and distributed tracing without a service mesh, but a service mesh moves these concerns out of the individual services and into a dedicated layer. On the other hand, service meshes are a relatively new technology that is still maturing. Deploying a service mesh adds complexity to the setup and configuration of the cluster. There may be performance implications, because requests now get routed through the service mesh proxy, and because extra services are now running on every node in the cluster. You should do thorough performance and load testing before deploying a service mesh in production.

API design

Designing microservices: API design

12/12/2017 • 10 min to read • Edit Online

Good API design is important in a microservices architecture, because all data exchange between services happens either through messages or API calls. APIs must be efficient to avoid creating chatty I/O. Because services are designed by teams working independently, APIs must have well-defined semantics and versioning schemes, so that updates don't break other services.



It's important to distinguish between two types of API:

- Public APIs that client applications call.
- Backend APIs that are used for interservice communication.

These two use cases have somewhat different requirements. A public API must be compatible with client applications, typically browser applications or native mobile applications. Most of the time, that means the public API will use REST over HTTP. For the backend APIs, however, you need to take network performance into account. Depending on the granularity of your services, interservice communication can result in a lot of network traffic. Services can quickly become I/O bound. For that reason, considerations such as serialization speed and payload size become more important. Some popular alternatives to using REST over HTTP include gRPC, Apache Avro, and Apache Thrift. These protocols support binary serialization and are generally more efficient than HTTP.

Considerations

Here are some things to think about when choosing how to implement an API.

REST vs RPC. Consider the tradeoffs between using a REST-style interface versus an RPC-style interface.

- REST models resources, which can be a natural way express your domain model. It defines a uniform interface based on HTTP verbs, which encourages evolvability. It has well-defined semantics in terms of idempotency, side effects, and response codes. And it enforces stateless communication, which improves scalability.
- RPC is more oriented around operations or commands. Because RPC interfaces look like local method calls, it may lead you to design overly chatty APIs. However, that doesn't mean RPC must be chatty. It just

means you need to use care when designing the interface.

For a RESTful interface, the most common choice is REST over HTTP using JSON. For an RPC-style interface, there are several popular frameworks, including gRPC, Apache Avro, and Apache Thrift.

Efficiency. Consider efficiency in terms of speed, memory, and payload size. Typically a gRPC-based interface is faster than REST over HTTP.

Interface definition language (IDL). An IDL is used to define the methods, parameters, and return values of an API. An IDL can be used to generate client code, serialization code, and API documentation. IDLs can also be consumed by API testing tools such as Postman. Frameworks such as gRPC, Avro, and Thrift define their own IDL specifications. REST over HTTP does not have a standard IDL format, but a common choice is OpenAPI (formerly Swagger). You can also create an HTTP REST API without using a formal definition language, but then you lose the benefits of code generation and testing.

Serialization. How are objects serialized over the wire? Options include text-based formats (primarily JSON) and binary formats such as protocol buffer. Binary formats are generally faster than text-based formats. However, JSON has advantages in terms of interoperability, because most languages and frameworks support JSON serialization. Some serialization formats require a fixed schema, and some require compiling a schema definition file. In that case, you'll need to incorporate this step into your build process.

Framework and language support. HTTP is supported in nearly every framework and language. gRPC, Avro, and Thrift all have libraries for C++, C#, Java, and Python. Thrift and gRPC also support Go.

Compatibility and interoperability. If you choose a protocol like gRPC, you may need a protocol translation layer between the public API and the back end. A gateway can perform that function. If you are using a service mesh, consider which protocols are compatible with the service mesh. For example, linkerd has built-in support for HTTP, Thrift, and gRPC.

Our baseline recommendation is to choose REST over HTTP unless you need the performance benefits of a binary protocol. REST over HTTP requires no special libraries. It creates minimal coupling, because callers don't need a client stub to communicate with the service. There is rich ecosystems of tools to support schema definitions, testing, and monitoring of RESTful HTTP endpoints. Finally, HTTP is compatible with browser clients, so you don't need a protocol translation layer between the client and the backend.

However, if you choose REST over HTTP, you should do performance and load testing early in the development process, to validate whether it performs well enough for your scenario.

RESTful API design

There are many resources for designing RESTful APIs. Here are some that you might find helpful:

- API design
- API implementation
- Microsoft REST API Guidelines

Here are some specific considerations to keep in mind.

- Watch out for APIs that leak internal implementation details or simply mirror an internal database schema. The API should model the domain. It's a contract between services, and ideally should only change when new functionality is added, not just because you refactored some code or normalized a database table.
- Different types of client, such as mobile application and desktop web browser, may require different payload sizes or interaction patterns. Consider using the Backends for Frontends pattern to create separate backends for each client, that expose an optimal interface for that client.
- For operations with side effects, consider making them idempotent and implementing them as PUT

methods. That will enable safe retries and can improve resiliency. The chapters Ingestion and workflow and Interservice communication discuss this issue in more detail.

- HTTP methods can have asynchronous semantics, where the method returns a response immediately, but the service carries out the operation asynchronously. In that case, the method should return an HTTP 202 response code, which indicates the request was accepted for processing, but the processing is not yet completed.

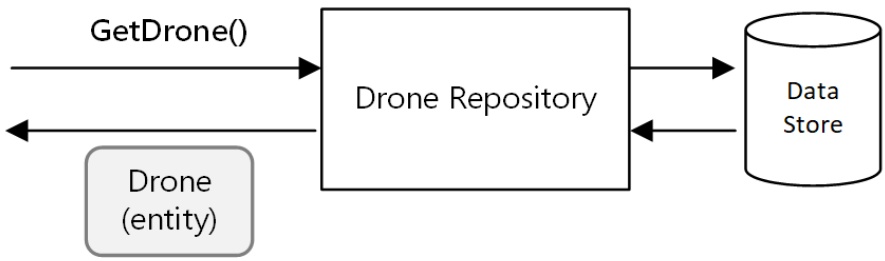
Mapping REST to DDD patterns

Patterns such as entity, aggregate, and value object are designed to place certain constraints on the objects in your domain model. In many discussions of DDD, the patterns are modeled using object-oriented (OO) language concepts like constructors or property getters and setters. For example, *value objects* are supposed to be immutable. In an OO programming language, you would enforce this by assigning the values in the constructor and making the properties read-only:

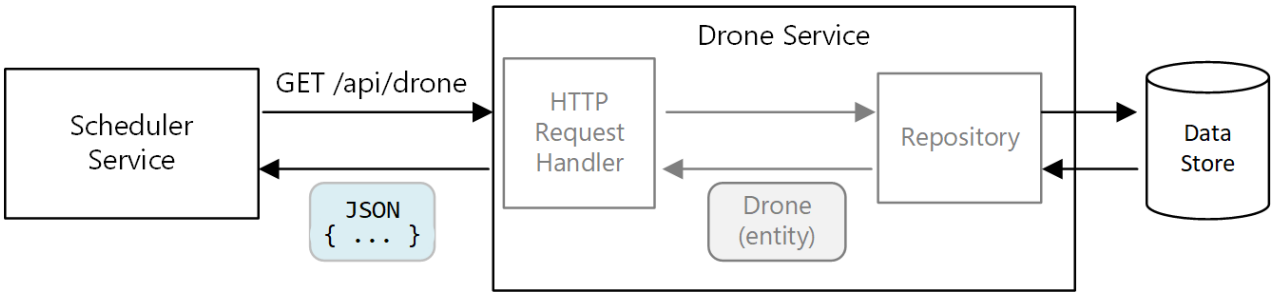
```
export class Location {
    readonly latitude: number;
    readonly longitude: number;

    constructor(latitude: number, longitude: number) {
        if (latitude < -90 || latitude > 90) {
            throw new RangeError('latitude must be between -90 and 90');
        }
        if (longitude < -180 || longitude > 180) {
            throw new RangeError('longitude must be between -180 and 180');
        }
        this.latitude = latitude;
        this.longitude = longitude;
    }
}
```

These sorts of coding practices are particularly important when building a traditional monolithic application. With a large code base, many subsystems might use the `Location` object, so it's important for the object to enforce correct behavior.

Another example is the Repository pattern, which ensures that other parts of the application do not make direct reads or writes to the data store:



In a microservices architecture, however, services don't share the same code base and don't share data stores. Instead, they communicate through APIs. Consider the case where the Scheduler service requests information about a drone from the Drone service. The Drone service has its internal model of a drone, expressed through code. But the Scheduler doesn't see that. Instead, it gets back a *representation* of the drone entity — perhaps a JSON object in an HTTP response.



The Scheduler service can't modify the Drone service's internal models, or write to the Drone service's data store. That means the code that implements the Drone service has a smaller exposed surface area, compared with code in a traditional monolith. If the Drone service defines a Location class, the scope of that class is limited — no other service will directly consume the class.

For these reasons, this guidance doesn't focus much on coding practices as they relate to the tactical DDD patterns. But it turns out that you can also model many of the DDD patterns through REST APIs.

For example:

- Aggregates map naturally to *resources* in REST. For example, the Delivery aggregate would be exposed as a resource by the Delivery API.
- Aggregates are consistency boundaries. Operations on aggregates should never leave an aggregate in an inconsistent state. Therefore, you should avoid creating APIs that allow a client to manipulate the internal state of an aggregate. Instead, favor coarse-grained APIs that expose aggregates as resources.
- Entities have unique identities. In REST, resources have unique identifiers in the form of URLs. Create resource URLs that correspond to an entity's domain identity. The mapping from URL to domain identity may be opaque to client.
- Child entities of an aggregate can be reached by navigating from the root entity. If you follow HATEOAS principles, child entities can be reached via links in the representation of the parent entity.
- Because value objects are immutable, updates are performed by replacing the entire value object. In REST, implement updates through PUT or PATCH requests.
- A repository lets clients query, add, or remove objects in a collection, abstracting the details of the underlying data store. In REST, a collection can be a distinct resource, with methods for querying the collection or adding new entities to the collection.

When you design your APIs, think about how they express the domain model, not just the data inside the model, but also the business operations and the constraints on the data.

DDD CONCEPT	REST EQUIVALENT	EXAMPLE
Aggregate	Resource	`{ "1":1234, "status":"pending"... }`
Identity	URL	`http://delivery-service/deliveries/1`
Child entities	Links	`{ "href": "/deliveries/1/confirmation" }`
Update value objects	PUT or PATCH	`PUT http://delivery-service/deliveries/1/dropoff`

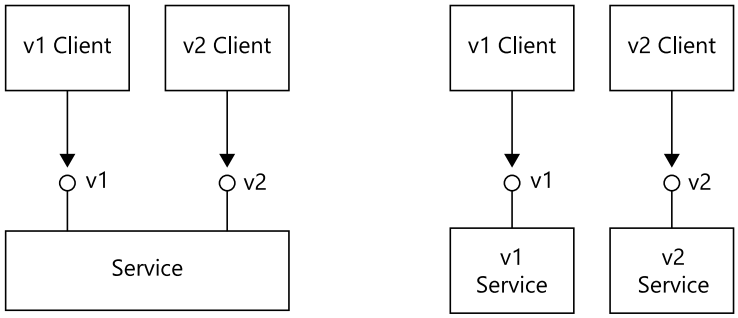
DDD CONCEPT	REST EQUIVALENT	EXAMPLE
Repository	Collection	`http://delivery-service/deliveries?status=pending`

API versioning

An API is a contract between a service and clients or consumers of that service. If an API changes, there is a risk of breaking clients that depend on the API, whether those are external clients or other microservices. Therefore, it's a good idea to minimize the number of API changes that you make. Often, changes in the underlying implementation don't require any changes to the API. Realistically, however, at some point you will want to add new features or new capabilities that require changing an existing API.

Whenever possible, make API changes backward compatible. For example, avoid removing a field from a model, because that can break clients that expect the field to be there. Adding a field does not break compatibility, because clients should ignore any fields they don't understand in a response. However, the service must handle the case where an older client omits the new field in a request.

Support versioning in your API contract. If you introduce a breaking API change, introduce a new API version. Continue to support the previous version, and let clients select which version to call. There are a couple of ways to do this. One is simply to expose both versions in the same service. Another option is to run two versions of the service side-by-side, and route requests to one or the other version, based on HTTP routing rules.



Service supports two versions *Side-by-side deployment*

There's a cost to supporting multiple versions, in terms of developer time, testing, and operational overhead. Therefore, it's good to deprecate old versions as quickly as possible. For internal APIs, the team that owns the API can work with other teams to help them migrate to the new version. This is when having a cross-team governance process is useful. For external (public) APIs, it can be harder to deprecate an API version, especially if the API is consumed by third parties or by native client applications.

When a service implementation changes, it's useful to tag the change with a version. The version provides important information when troubleshooting errors. It can be very helpful for root cause analysis to know exactly which version of the service was called. Consider using semantic versioning for service versions. Semantic versioning uses a *MAJOR.MINOR.PATCH* format. However, clients should only select an API by the major version number, or possibly the minor version if there are significant (but non-breaking) changes between minor versions. In other words, it's reasonable for clients to select between version 1 and version 2 of an API, but not to select version 2.1.3. If you allow that level of granularity, you risk having to support a proliferation of versions.

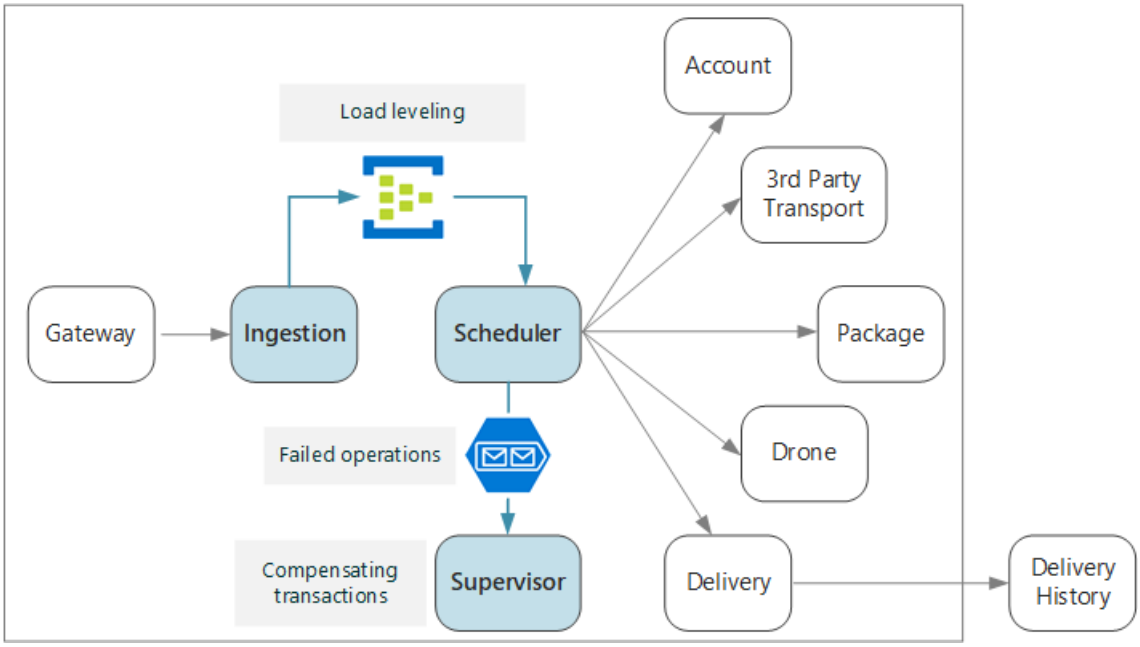
For further discussion of API versioning, see Versioning a RESTful web API.

Ingestion and workflow

Designing microservices: Ingestion and workflow

12/13/2017 • 17 min to read • Edit Online

Microservices often have a workflow that spans multiple services for a single transaction. The workflow must be reliable; it can't lose transactions or leave them in a partially completed state. It's also critical to control the ingestion rate of incoming requests. With many small services communicating with each other, a burst of incoming requests can overwhelm the interservice communication.



The drone delivery workflow

In the Drone Delivery application, the following operations must be performed to schedule a delivery:

1. Check the status of the customer's account (Account service).
2. Create a new package entity (Package service).
3. Check whether any third-party transportation is required for this delivery, based on the pickup and delivery locations (Third-party Transportation service).
4. Schedule a drone for pickup (Drone service).
5. Create a new delivery entity (Delivery service).

This is the core of the entire application, so the end-to-end process must be performant as well as reliable. Some particular challenges must be addressed:

- **Load leveling**. Too many client requests can overwhelm the system with interservice network traffic. It can also overwhelm backend dependencies such as storage or remote services. These may react by throttling the services calling them, creating backpressure in the system. Therefore, it's important to load level the requests coming into the system, by putting them into a buffer or queue for processing.
- **Guaranteed delivery**. To avoid dropping any client requests, the ingestion component must guarantee at-least-once delivery of messages.
- **Error handling**. If any of the services returns an error code or experiences a non-transient failure, the delivery cannot be scheduled. An error code might indicate an expected error condition (for example, the customer's account is suspended) or an unexpected server error (HTTP 5xx). A service might also be

unavailable, causing the network call to time out.

First we'll look at the ingestion side of the equation — how the system can ingest incoming user requests at high throughput. Then we'll consider how the drone delivery application can implement a reliable workflow. It turns out that the design of the ingestion subsystem affects the workflow backend.

Ingestion

Based on business requirements, the development team identified the following non-functional requirements for ingestion:

- Sustained throughput of 10K requests/sec.
- Able to handle spikes of up to 50K/sec without dropping client requests or timing out.
- Less than 500ms latency in the 99th percentile.

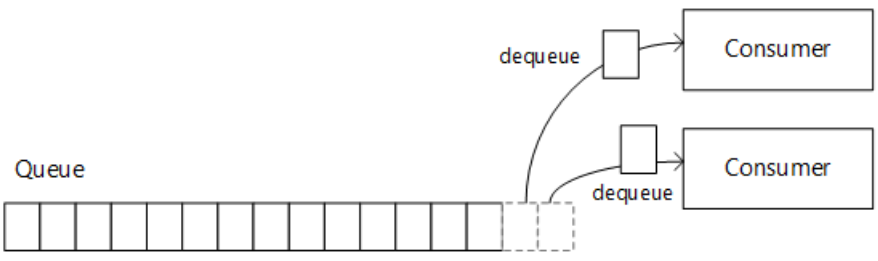
The requirement to handle occasional spikes in traffic presents a design challenge. In theory, the system could be scaled out to handle the maximum expected traffic. However, provisioning that many resources would be very inefficient. Most of the time, the application will not need that much capacity, so there would be idle cores, costing money without adding value.

A better approach is to put the incoming requests into a buffer, and let the buffer act as a load leveler. With this design, the Ingestion service must be able to handle the maximum ingestion rate over short periods, but the backend services only need to handle the maximum sustained load. By buffering at the front end, the backend services shouldn't need to handle large spikes in traffic. At the scale required for the Drone Delivery application, Azure Event Hubs is a good choice for load leveling. Event Hubs offers low latency and high throughput, and is a cost effective solution at high ingestion volumes.

For our testing, we used a Standard tier event hub with 32 partitions and 100 throughput units. We observed about 32K events / second ingestion, with latency around 90ms. Currently the default limit is 20 throughput units, but Azure customers can request additional throughput units by filing a support request. See Event Hubs quotas for more information. As with all performance metrics, many factors can affect performance, such as message payload size, so don't interpret these numbers as a benchmark. If more throughput is needed, the Ingestion service can shard across more than one event hub. For even higher throughput rates, Event Hubs Dedicated offers single-tenant deployments that can ingress over 2 million events per second.

It's important to understand how Event Hubs can achieve such high throughput, because that affects how a client should consume messages from Event Hubs. Event Hubs does not implement a *queue*. Rather, it implements an *event stream*.

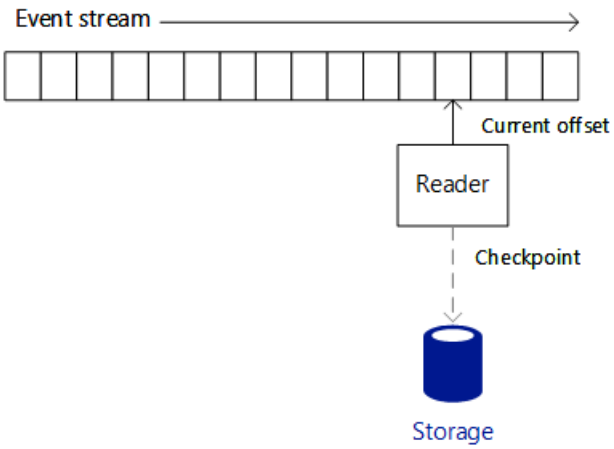
With a queue, an individual consumer can remove a message from the queue, and the next consumer won't see that message. Queues therefore allow you to use a Competing Consumers pattern to process messages in parallel and improve scalability. For greater resiliency, the consumer holds a lock on the message and releases the lock when it's done processing the message. If the consumer fails — for example, the node it's running on crashes — the lock times out and the message goes back onto the queue.



Event Hubs, on the other hand, uses streaming semantics. Consumers read the stream independently at their own pace. Each consumer is responsible for keeping track of its current position in the stream. A consumer should write its current position to persistent storage at some predefined interval. That way, if the consumer experiences

a fault (for example, the consumer crashes, or the host fails), then a new instance can resume reading the stream from the last recorded position. This process is called *checkpointing*.

For performance reasons, a consumer generally doesn't checkpoint after each message. Instead, it checkpoints at some fixed interval, for example after processing *n* messages, or every *n* seconds. As a consequence, if a consumer fails, some events may get processed twice, because a new instance always picks up from the last checkpoint. There is a tradeoff: Frequent checkpoints can hurt performance, but sparse checkpoints mean you will replay more events after a failure.



Event Hubs is not designed for competing consumers. Although multiple consumers can read a stream, each traverses the stream independently. Instead, Event Hubs uses a partitioned consumer pattern. An event hub has up to 32 partitions. Horizontal scale is achieved by assigning a separate consumer to each partition.

What does this mean for the drone delivery workflow? To get the full benefit of Event Hubs, the Delivery Scheduler cannot wait for each message to be processed before moving onto the next. If it does that, it will spend most of its time waiting for network calls to complete. Instead, it needs to process batches of messages in parallel, using asynchronous calls to the backend services. As we'll see, choosing the right checkpointing strategy is also important.

Workflow

We looked at three options for reading and processing the messages: Event Processor Host, Service Bus queues, and the IoTHub React library. We chose IoTHub React, but to understand why, it helps to start with Event Processor Host.

Event Processor Host

Event Processor Host is designed for message batching. The application implements the `IEventProcessor` interface, and the Processor Host creates one event processor instance for each partition in the event hub. The Event Processor Host then calls each event processor's `ProcessEventsAsync` method with batches of event messages. The application controls when to checkpoint inside the `ProcessEventsAsync` method, and the Event Processor Host writes the checkpoints to Azure storage.

Within a partition, Event Processor Host waits for `ProcessEventsAsync` to return before calling again with the next batch. This approach simplifies the programming model, because your event processing code doesn't need to be reentrant. However, it also means that the event processor handles one batch at a time, and this gates the speed at which the Processor Host can pump messages.

> **NOTE**
>
> The Processor Host doesn't actually *wait* in the sense of blocking a thread. The `ProcessEventsAsync` method is asynchronous, so the Processor Host can do other work while the method is completing. But it won't deliver another batch of messages for that partition until the method returns.

In the drone application, a batch of messages can be processed in parallel. But waiting for the whole batch to complete can still cause a bottleneck. Processing can only be as fast as the slowest message within a batch. Any variation in response times can create a "long tail," where a few slow responses drag down the entire system. Our performance tests showed that we did not achieve our target throughput using this approach. This does *not* mean that you should avoid using Event Processor Host. But for high throughput, avoid doing any long-running tasks inside the `ProcesssEventsAsync` method. Process each batch quickly.

IotHub React

IotHub React is an Akka Streams library for reading events from Event Hub. Akka Streams is a stream-based programming framework that implements the Reactive Streams specification. It provides a way to build efficient streaming pipelines, where all streaming operations are performed asynchronously, and the pipeline gracefully handles backpressure. Backpressure occurs when an event source produces events at a faster rate than the downstream consumers can receive them — which is exactly the situation when the drone delivery system has a spike in traffic. If backend services go slower, IoTHub React will slow down. If capacity is increased, IoTHub React will push more messages through the pipeline.

Akka Streams is also a very natural programming model for streaming events from Event Hubs. Instead of looping through a batch of events, you define a set of operations that will be applied to each event, and let Akka Streams handle the streaming. Akka Streams defines a streaming pipeline in terms of *Sources*, *Flows*, and *Sinks*. A source generates an output stream, a flow processes an input stream and produces an output stream, and a sink consumes a stream without producing any output.

Here is the code in the Scheduler service that sets up the Akka Streams pipeline:

```
IoTHub iotHub = new IoTHub();
Source<MessageFromDevice, NotUsed> messages = iotHub.source(options);

messages.map(msg -> DeliveryRequestEventProcessor.parseDeliveryRequest(msg))
        .filter(ad -> ad.getDelivery() != null).via(deliveryProcessor()).to(iotHub.checkpointSink())
        .run(streamMaterializer);
```

This code configures Event Hubs as a source. The `map` statement deserializes each event message into a Java class that represents a delivery request. The `filter` statement removes any `null` objects from the stream; this guards against the case where a message can't be deserialized. The `via` statement joins the source to a flow that processes each delivery request. The `to` method joins the flow to the checkpoint sink, which is built into IoTHub React.

IoTHub React uses a different checkpointing strategy than Event Host Processor. Checkpoints are written by the checkpoint sink, which is the terminating stage in the pipeline. The design of Akka Streams allows the pipeline to continue streaming data while the sink is writing the checkpoint. That means the upstream processing stages don't need to wait for checkpointing to happen. You can configure checkpointing to occur after a timeout or after a certain number of messages have been processed.

The `deliveryProcessor` method creates the Akka Streams flow:

```
private static Flow<AkkaDelivery, MessageFromDevice, NotUsed> deliveryProcessor() {
    return Flow.of(AkkaDelivery.class).map(delivery -> {
        CompletableFuture<DeliverySchedule> completableSchedule = DeliveryRequestEventProcessor
                .processDeliveryRequestAsync(delivery.getDelivery(),
                        delivery.getMessageFromDevice().properties());

        completableSchedule.whenComplete((deliverySchedule,error) -> {
            if (error!=null){
                Log.info("failed delivery" + error.getStackTrace());
            }
            else{
                Log.info("Completed Delivery",deliverySchedule.toString());
            }

        });
        completableSchedule = null;
        return delivery.getMessageFromDevice();
    });
}
```

The flow calls a static `processDeliveryRequestAsync` method that does the actual work of processing each message.

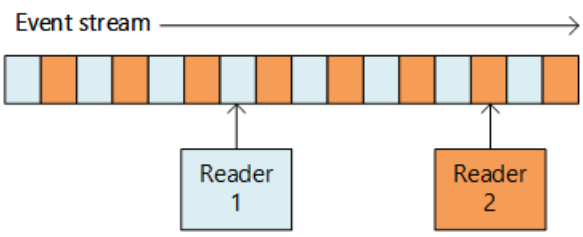
Scaling with IoTHub React

The Scheduler service is designed so that each container instance reads from a single partition. For example, if the Event Hub has 32 partitions, the Scheduler service is deployed with 32 replicas. This allows for a lot of flexibility in terms of horizontal scaling.

Depending on the size of the cluster, a node in the cluster might have more than one Scheduler service pod running on it. But if the Scheduler service needs more resources, the cluster can be scaled out, in order to distribute the pods across more nodes. Our performance tests showed that the Scheduler service is memory- and thread-bound, so performance depended greatly on the VM size and the number of pods per node.

Each instance needs to know which Event Hubs partition to read from. To configure the partition number, we took advantage of the StatefulSet resource type in Kubernetes. Pods in a StatefulSet have a persistent identifier that includes a numeric index. Specifically, the pod name is `<statefulset name>-<index>`, and this value is available to the container through the Kubernetes Downward API. At run time, the Scheduler services reads the pod name and uses the pod index as the partition ID.

If you needed to scale out the Scheduler service even further, you could assign more than one pod per event hub partition, so that multiple pods are reading each partition. However, in that case, each instance would read all of the events in the assigned partition. To avoid duplicate processing, you would need to use a hashing algorithm, so that each instance skips over a portion of the messages. That way, multiple readers can consume the stream, but every message is processed by only one instance.



Service Bus queues

A third option that we considered was to copy messages from Event Hubs into a Service Bus queue, and then have the Scheduler service read the messages from Service Bus. It might seem strange to writing the incoming requests into Event Hubs only to copy them in Service Bus. However, the idea was to leverage the different strengths of each service: Use Event Hubs to absorb spikes of heavy traffic, while taking advantage of the queue semantics in Service Bus to process the workload with a competing consumers pattern. Remember that our target

for sustained throughput is less than our expected peak load, so processing the Service Bus queue would not need to be as fast the message ingestion.

With this approach, our proof-of-concept implementation achieved about 4K operations per second. These tests used mock backend services that did not do any real work, but simply added a fixed amount of latency per service. Note that our performance numbers were much less than the theoretical maximum for Service Bus. Possible reasons for the discrepancy include:

- Not having optimal values for various client parameters, such as the connection pool limit, the degree of parallelization, the prefetch count, and the batch size.
- Network I/O bottlenecks.
- Use of PeekLock mode rather than ReceiveAndDelete, which was needed to ensure at-least-once delivery of messages.

Further performance tests might have discovered the root cause and allowed us to resolve these issues. However, IotHub React met our performance target, so we chose that option. That said, Service Bus is a viable option for this scenario.

Handling failures

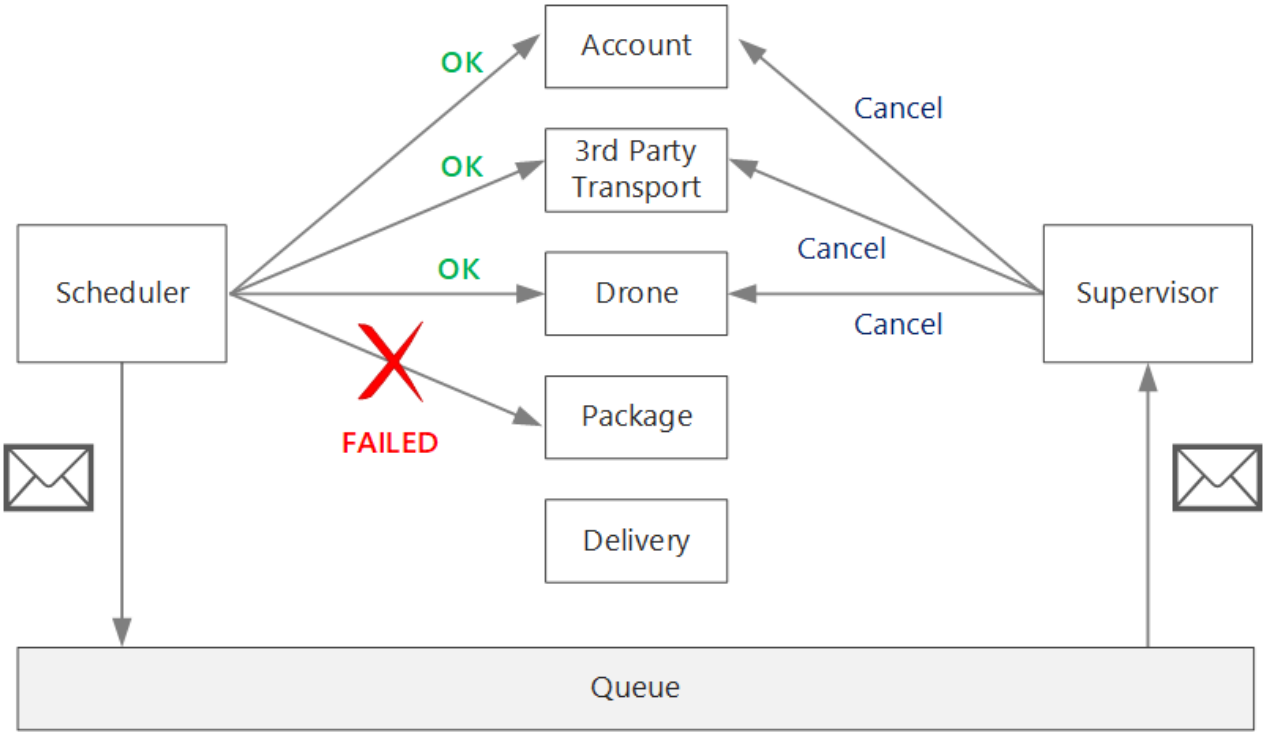
There are three general classes of failure to consider.

1. A downstream service may have a non-transient failure, which is any failure that's unlikely to go away by itself. Non-transient failures include normal error conditions, such as invalid input to a method. They also include unhandled exceptions in application code or a process crashing. If this type of error occurs, the entire business transaction must be marked as a failure. It may be necessary to undo other steps in the same transaction that already succeeded. (See Compensating Transactions, below.)
2. A downstream service may experience a transient failure such as a network timeout. These errors can often be resolved simply by retrying the call. If the operation still fails after a certain number of attempts, it's considered a non-transient failure.
3. The Scheduler service itself might fault (for example, because a node crashes). In that case, Kubernetes will bring up a new instance of the service. However, any transactions that were already in progress must be resumed.

Compensating transactions

If a non-transient failure happens, the current transaction might be in a *partially failed* state, where one or more steps already completed successfully. For example, if the Drone service already scheduled a drone, the drone must be canceled. In that case, the application needs to undo the steps that succeeded, by using a Compensating Transaction. In some cases, this must be done by an external system or even by a manual process.

If the logic for compensating transactions is complex, consider creating a separate service that is responsible for this process. In the Drone Delivery application, the Scheduler service puts failed operations onto a dedicated queue. A separate microservice, called the Supervisor, reads from this queue and calls a cancellation API on the services that need to compensate. This is a variation of the Scheduler Agent Supervisor pattern. The Supervisor service might take other actions as well, such as notify the user by text or email, or send an alert to an operations dashboard.



Idempotent vs non-idempotent operations

To avoid losing any requests, the Scheduler service must guarantee that all messages are processed at least once. Event Hubs can guarantee at-least-once delivery if the client checkpoints correctly.

If the Scheduler service crashes, it may be in the middle of processing one or more client requests. Those messages will be picked up by another instance of the Scheduler and reprocessed. What happens if a request is processed twice? It's important to avoid duplicating any work. After all, we don't want the system to send two drones for the same package.

One approach is to design all operations to be idempotent. An operation is idempotent if it can be called multiple times without producing additional side-effects after the first call. In other words, a client can invoke the operation once, twice, or many times, and the result will be the same. Essentially, the service should ignore duplicate calls. For a method with side effects to be idempotent, the service must be able to detect duplicate calls. For example, you can have the caller assign the ID, rather than having the service generate a new ID. The service can then check for duplicate IDs.

> **NOTE**
>
> The HTTP specification states that GET, PUT, and DELETE methods must be idempotent. POST methods are not guaranteed to be idempotent. If a POST method creates a new resource, there is generally no guarantee that this operation is idempotent.

It's not always straightforward to write idempotent method. Another option is for the Scheduler to track the progress of every transaction in a durable store. Whenever it processes a message, it would look up the state in the durable store. After each step, it would write the result to the store. There may be performance implications to this approach.

Example: Idempotent operations

The HTTP specification states that PUT methods must be idempotent. The specification defines idempotent this way:

> A request method is considered "idempotent" if the intended effect on the server of multiple identical

> requests with that method is the same as the effect for a single such request. (RFC 7231)

It's important to understand the difference between PUT and POST semantics when creating a new entity. In both cases, the client sends a representation of an entity in the request body. But the meaning of the URI is different.

- For a POST method, the URI represents a parent resource of the new entity, such as a collection. For example, to create a new delivery, the URI might be `/api/deliveries`. The server creates the entity and assigns it a new URI, such as `/api/deliveries/39660`. This URI is returned in the Location header of the response. Each time the client sends a request, the server will create a new entity with a new URI.
- For a PUT method, the URI identifies the entity. If there already exists an entity with that URI, the server replaces the existing entity with the version in the request. If no entity exists with that URI, the server creates one. For example, suppose the client sends a PUT request to `api/deliveries/39660`. Assuming there is no delivery with that URI, the server creates a new one. Now if the client sends the same request again, the server will replace the existing entity.

Here is the Delivery service's implementation of the PUT method.

```
[HttpPut("{id}")]
[ProducesResponseType(typeof(Delivery), 201)]
[ProducesResponseType(typeof(void), 204)]
public async Task<IActionResult> Put([FromBody]Delivery delivery, string id)
{
    logger.LogInformation("In Put action with delivery {Id}: {@DeliveryInfo}", id, delivery.ToLogInfo());
    try
    {
        var internalDelivery = delivery.ToInternal();

        // Create the new delivery entity.
        await deliveryRepository.CreateAsync(internalDelivery);

        // Create a delivery status event.
        var deliveryStatusEvent = new DeliveryStatusEvent { DeliveryId = delivery.Id, Stage =
DeliveryEventType.Created };
        await deliveryStatusEventRepository.AddAsync(deliveryStatusEvent);

        // Return HTTP 201 (Created)
        return CreatedAtRoute("GetDelivery", new { id= delivery.Id }, delivery);
    }
    catch (DuplicateResourceException)
    {
        // This method is mainly used to create deliveries. If the delivery already exists then update it.
        logger.LogInformation("Updating resource with delivery id: {DeliveryId}", id);

        var internalDelivery = delivery.ToInternal();
        await deliveryRepository.UpdateAsync(id, internalDelivery);

        // Return HTTP 204 (No Content)
        return NoContent();
    }
}
```

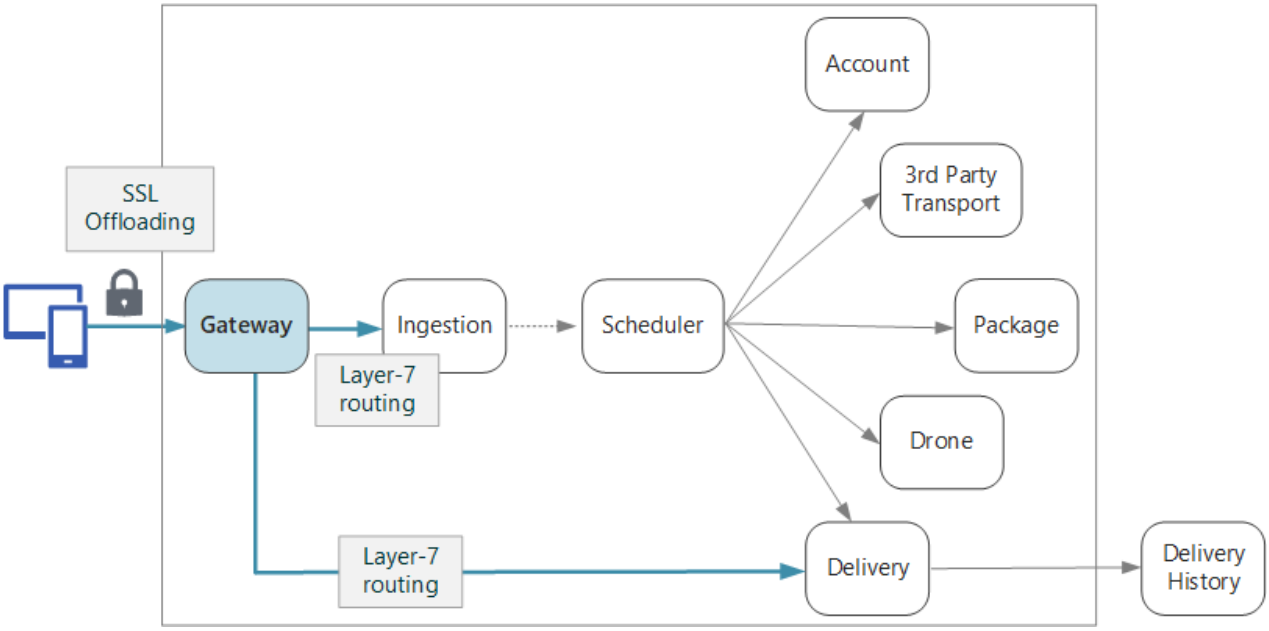
It's expected that most requests will create a new entity, so the method optimistically calls `CreateAsync` on the repository object, and then handles any duplicate-resource exceptions by updating the resource instead.

API gateways

Designing microservices: API gateways

12/29/2017 • 7 min to read • Edit Online

In a microservices architecture, a client might interact with more than one front-end service. Given this fact, how does a client know what endpoints to call? What happens when new services are introduced, or existing services are refactored? How do services handle SSL termination, authentication, and other concerns? An *API gateway* can help to address these challenges.



What is an API gateway?

An API gateway sits between clients and services. It acts as a reverse proxy, routing requests from clients to services. It may also perform various cross-cutting tasks such as authentication, SSL termination, and rate limiting. If you don't deploy a gateway, clients must send requests directly to front-end services. However, there are some potential problems with exposing services directly to clients:

- It can result in complex client code. The client must keep track of multiple endpoints, and handle failures in a resilient way.
- It creates coupling between the client and the backend. The client needs to know how the individual services are decomposed. That makes it harder to maintain the client and also harder to refactor services.
- A single operation might require calls to multiple services. That can result in multiple network round trips between the client and the server, adding significant latency.
- Each public-facing service must handle concerns such as authentication, SSL, and client rate limiting.
- Services must expose a client-friendly protocol such as HTTP or WebSocket. This limits the choice of communication protocols.
- Services with public endpoints are a potential attack surface, and must be hardened.

A gateway helps to address these issues by decoupling clients from services. Gateways can perform a number of different functions, and you may not need all of them. The functions can be grouped into the following design patterns:

Gateway Routing. Use the gateway as a reverse proxy to route requests to one or more backend services, using layer 7 routing. The gateway provides a single endpoint for clients, and helps to decouple clients from services.

Gateway Aggregation. Use the gateway to aggregate multiple individual requests into a single request. This pattern applies when a single operation requires calls to multiple backend services. The client sends one request to the gateway. The gateway dispatches requests to the various backend services, and then aggregates the results and sends them back to the client. This helps to reduce chattiness between the client and the backend.

Gateway Offloading. Use the gateway to offload functionality from individual services to the gateway, particularly cross-cutting concerns. It can be useful to consolidate these functions into one place, rather than making every service responsible for implementing them. This is particularly true for features that requires specialized skills to implement correctly, such as authentication and authorization.

Here are some examples of functionality that could be offloaded to a gateway:

- SSL termination
- Authentication
- IP whitelisting
- Client rate limiting (throttling)
- Logging and monitoring
- Response caching
- Web application firewall
- GZIP compression
- Servicing static content

Choosing a gateway technology

Here are some options for implementing an API gateway in your application.

- **Reverse proxy server**. Nginx and HAProxy are popular reverse proxy servers that support features such as load balancing, SSL, and layer 7 routing. They are both free, open-source products, with paid editions that provide additional features and support options. Nginx and HAProxy are both mature products with rich feature sets and high performance. You can extend them with third-party modules or by writing custom scripts in Lua. Nginx also supports a JavaScript-based scripting module called NginScript.

- **Service mesh ingress controller**. If you are using a service mesh such as linkerd or Istio, consider the features that are provided by the ingress controller for that service mesh. For example, the Istio ingress controller supports layer 7 routing, HTTP redirects, retries, and other features.

- Azure Application Gateway. Application Gateway is a managed load balancing service that can perform layer-7 routing and SSL termination. It also provides a web application firewall (WAF).

- Azure API Management. API Management is a turnkey solution for publishing APIs to external and internal customers. It provides features that are useful for managing a public-facing API, including rate limiting, IP white listing, and authentication using Azure Active Directory or other identity providers. API Management doesn't perform any load balancing, so it should be used in conjunction with a load balancer such as Application Gateway or a reverse proxy.

When choosing a gateway technology, consider the following:

Features. The options listed above all support layer 7 routing, but support for other features will vary. Depending on the features that you need, you might deploy more than one gateway.

Deployment. Azure Application Gateway and API Management are managed services. Nginx and HAProxy will typically run in containers inside the cluster, but can also be deployed to dedicated VMs outside of the cluster. This isolates the gateway from the rest of the workload, but incurs higher management overhead.

Management. When services are updated or new services are added, the gateway routing rules may need to be updated. Consider how this process will be managed. Similar considerations apply to managing SSL certificates,

IP whitelists, and other aspects of configuration.

Deployment considerations

Deploying Nginx or HAProxy to Kubernetes

You can deploy Nginx or HAProxy to Kubernetes as a ReplicaSet or DaemonSet that specifies the Nginx or HAProxy container image. Use a ConfigMap to store the configuration file for the proxy, and mount the ConfigMap as a volume. Create a service of type LoadBalancer to expose the gateway through an Azure Load Balancer.

An alternative is to create an Ingress Controller. An Ingress Controller is a Kubernetes resource that deploys a load balancer or reverse proxy server. Several implementations exist, including Nginx and HAProxy. A separate resource called an Ingress defines settings for the Ingress Controller, such as routing rules and TLS certificates. That way, you don't need to manage complex configuration files that are specific to a particular proxy server technology. Ingress Controllers are still a beta feature of Kubernetes at the time of this writing, and the feature will continue to evolve.

The gateway is a potential bottleneck or single point of failure in the system, so always deploy at least two replicas for high availability. You may need to scale out the replicas further, depending on the load.

Also consider running the gateway on a dedicated set of nodes in the cluster. Benefits to this approach include:

- Isolation. All inbound traffic goes to a fixed set of nodes, which can be isolated from backend services.
- Stable configuration. If the gateway is misconfigured, the entire application may become unavailable.
- Performance. You may want to use a specific VM configuration for the gateway for performance reasons.

Azure Application Gateway

To connect Application Gateway to a Kubernetes cluster in Azure:

1. Create an empty subnet in the cluster VNet.
2. Deploy Application Gateway.
3. Create a Kubernetes service with type=NodePort. This exposes the service on each node so that it can be reached from outside the cluster. It does not create a load balancer.
4. Get the assigned port number for the service.
5. Add an Application Gateway rule where:
   - The backend pool contains the agent VMs.
   - The HTTP setting specifies the service port number.
   - The gateway listener listens on ports 80/443

Set the instance count to 2 or more for high availability.

Azure API Management

To connect API Management to a Kubernetes cluster in Azure:

1. Create an empty subnet in the cluster VNet.
2. Deploy API Management to that subnet.
3. Create a Kubernetes service of type LoadBalancer. Use the internal load balancer annotation to create an internal load balancer, instead of an Internet-facing load balancer, which is the default.
4. Find the private IP of the internal load balancer, using kubectl or the Azure CLI.
5. Use API Management to create an API that directs to the private IP address of the load balancer.

Consider combining API Management with a reverse proxy, whether Nginx, HAProxy, or Azure Application Gateway. For information about using API Management with Application Gateway, see Integrate API

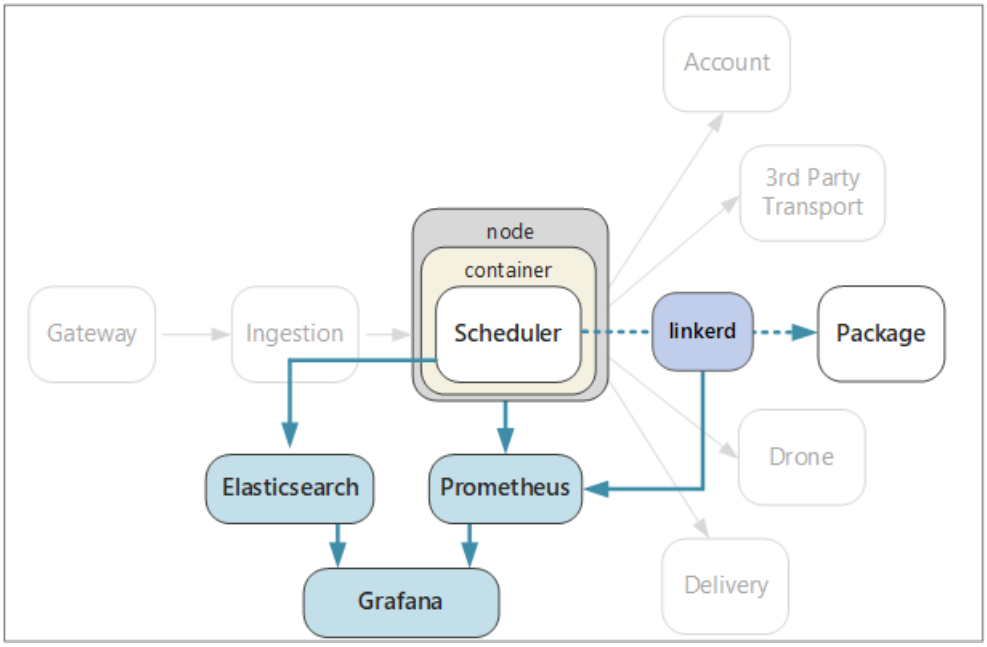
Management in an internal VNET with Application Gateway.

Logging and monitoring

Designing microservices: Logging and monitoring

12/12/2017 • 13 min to read • Edit Online

In any complex application, at some point something will go wrong. In a microservices application, you need to track what's happening across dozens or even hundreds of services. Logging and monitoring are critically important to give you a holistic view of the system.



In a microservices architecture, it can be especially challenging to pinpoint the exact cause of errors or performance bottlenecks. A single user operation might span multiple services. Services may hit network I/O limits inside the cluster. A chain of calls across services may cause backpressure in the system, resulting in high latency or cascading failures. Moreover, you generally don't know which node a particular container will run in. Containers placed on the same node may be competing for limited CPU or memory.

To make sense of what's happening, the application must emit telemetry events. You can categorize these into metrics and text-based logs.

Metrics are numerical values that can be analyzed. You can use them to observe the system in real time (or close to real time), or to analyze performance trends over time. Metrics include:

- Node-level system metrics, including CPU, memory, network, disk, and file system usage. System metrics help you to understand resource allocation for each node in the cluster, and troubleshoot outliers.
- Kubernetes metrics. Because services run in containers, you need to collect metrics at the container level, not just at the VM level. In Kubernetes, cAdvisor (Container Advisor) is the agent that collects statistics about the CPU, memory, file system, and network resources used by each container. The kubelet daemon collects resource statistics from cAdvisor and exposes them through a REST API.
- Application metrics. This includes any metrics that are relevant to understanding the behavior of a service. Examples include the number of queued inbound HTTP requests, request latency, message queue length, or number of transactions processed per second.
- Dependent service metrics. Services inside the cluster may call external services that are outside the cluster, such as managed PaaS services. You can monitor Azure services by using Azure Monitor. Third-party services may or may not provide any metrics. If not, you'll have to rely on your own application metrics to track statistics for latency and error rate.

Logs are records of events that occur while the application is running. They include things like application logs (trace statements) or web server logs. Logs are primarily useful for forensics and root cause analysis.

Considerations

The article Monitoring and diagnostics describes general best practices for monitoring an application. Here are some particular things to think about in the context of a microservices architecture.

Configuration and management. Will you use a managed service for logging and monitoring, or deploy logging and monitoring components as containers inside the cluster? For more discussion of these options, see the section Technology Options below.

Ingestion rate. What is the throughput at which the system can ingest telemetry events? What happens if that rate is exceeded? For example, the system may throttle clients, in which case telemetry data is lost, or it may downsample the data. Sometimes you can mitigate this problem by reducing the amount of data that you collect:

- Aggregate metrics by calculating statistics, such as average and standard deviation, and send that statistical data to the monitoring system.
- Downsample the data — that is, process only a percentage of the events.
- Batch the data to reduce the number of network calls to the monitoring service.

Cost. The cost of ingesting and storing telemetry data may be high, especially at high volumes. In some cases it could even exceed the cost of running the application. In that case, you may need to reduce the volume of telemetry by aggregating, downsampling, or batching the data, as described above.

Data fidelity. How accurate are the metrics? Averages can hide outliers, especially at scale. Also, if the sampling rate is too low, it can smooth out fluctuations in the data. It may appear that all requests have about the same end-to-end latency, when in fact a significant fraction of requests are taking much longer.

Latency. To enable real-time monitoring and alerts, telemetry data should be available quickly. How "real-time" is the data that appears on the monitoring dashboard? A few seconds old? More than a minute?

Storage. For logs, it may be most efficient to write the log events to ephemeral storage in the cluster, and configure an agent to ship the log files to more persistent storage. Data should eventually be moved to long-term storage so that it's available for retrospective analysis. A microservices architecture can generate a large volume of telemetry data, so the cost of storing that data is an important consideration. Also consider how you will query the data.

Dashboard and visualization. Do you get a holistic view of the system, across all of the services, both within the cluster and external services? If you are writing telemetry data and logs to more than one location, can the dashboard show all of them and correlate? The monitoring dashboard should show at least the following information:

- Overall resource allocation for capacity and growth. This includes the number of containers, file system metrics, network, and core allocation.
- Container metrics correlated at the service level.
- System metrics correlated with containers.
- Service errors and outliers.

Distributed tracing

As mentioned, one challenge in microservices is understanding the flow of events across services. A single operation or transaction may involve calls to multiple services. To reconstruct the entire sequence of steps, each service should propagate a *correlation ID* that acts as a unique identifier for that operation. The correlation ID enables distributed tracing across services.

The first service that receives a client request should generate the correlation ID. If the service makes an HTTP call to another service, it puts the correlation ID in a request header. Similarly, if the service sends an asynchronous message, it puts the correlation ID into the message. Downstream services continue to propagate the correlation ID, so that it flows through the entire system. In addition, all code that writes application metrics or log events should include the correlation ID.

When service calls are correlated, you can calculate operational metrics such as the end-to-end latency for a complete transaction, the number of successful transactions per second, and the percentage of failed transactions. Including correlation IDs in application logs makes it possible to perform root cause analysis. If an operation fails, you can find the log statements for all of the service calls that were part of the same operation.

Here are some considerations when implementing distributed tracing:

- There is currently no standard HTTP header for correlation IDs. Your team should standardize on a custom header value. The choice may be decided by your logging/monitoring framework or choice of service mesh.
- For asynchronous messages, if your messaging infrastructure supports adding metadata to messages, you should include the correlation ID as metadata. Otherwise, include it as part of the message schema.
- Rather than a single opaque identifier, you might send a *correlation context* that includes richer information, such as caller-callee relationships.
- The Azure Application Insights SDK automatically injects correlation context into HTTP headers, and includes the correlation ID in Application Insights logs. If you decide to use the correlation features built into Application Insights, some services may still need to explicitly propagate the correlation headers, depending on the libraries being used. For more information, see Telemetry correlation in Application Insights.
- If you are using Istio or linkerd as a service mesh, these technologies automatically generate correlation headers when HTTP calls are routed through the service mesh proxies. Services should forward the relevant headers.
  - Istio: Distributed Request Tracing
  - linkerd: Context Headers
- Consider how you will aggregate logs. You may want to standardize across teams on how to include correlation IDs in logs. Use a structured or semi-structured format, such as JSON, and define a common field to hold the correlation ID.

Technology options

Application Insights is a managed service in Azure that ingests and stores telemetry data, and provides tools for analyzing and searching the data. To use Application Insights, you install an instrumentation package in your application. This package monitors the app and sends telemetry data to the Application Insights service. It can also pull telemetry data from the host environment. Application Insights provides built-in correlation and dependency tracking. It lets you track system metrics, application metrics, and Azure service metrics, all in one place.

Be aware that Application Insights throttles if the data rate exceeds a maximum limit; for details, see Application Insights limits. A single operation may generate several telemetry events, so if the application experiences a high volume of traffic, it is likely to get throttled. To mitigate this problem, you can perform sampling to reduce the telemetry traffic. The tradeoff is that your metrics will be less precise. For more information, see Sampling in Application Insights. You can also reduce the data volume by pre-aggregating metrics — that is, calculating statistical values such as average and standard deviation, and sending those values instead of the raw telemetry. The following blog post describes an approach to using Application Insights at scale: Azure Monitoring and Analytics at Scale.

In addition, make sure that you understand the pricing model for Application Insights, because you are charged based on data volume. For more information, see Manage pricing and data volume in Application Insights. If your application generates a large volume of telemetry, and you don't wish to perform sampling or aggregation of the data, then Application Insights may not be the appropriate choice.

If Application Insights doesn't meet your requirements, here are some suggested approaches that use popular open-source technologies.

For system and container metrics, consider exporting metrics to a time-series database such as **Prometheus** or **InfluxDB** running in the cluster.

- InfluxDB is a push-based system. An agent needs to push the metrics. You can use Heapster, which is a service that collects cluster-wide metrics from kubelet, aggregates the data, and pushes it to InfluxDB or other time-series storage solution. Azure Container Service deploys Heapster as part of the cluster setup. Another option is Telegraf, which is an agent for collecting and reporting metrics.
- Prometheus is a pull-based system. It periodically scrapes metrics from configured locations. Prometheus can scrape metrics generated by cAdvisor or kube-state-metrics. kube-state-metrics is a service that collects metrics from the Kubernetes API server and makes them available to Prometheus (or a scraper that is compatible with a Prometheus client endpoint). Whereas Heapster aggregates metrics that Kubernetes generates and forwards them to a sink, kube-state-metrics generates its own metrics and makes them available through an endpoint for scraping. For system metrics, use Node exporter, which is a Prometheus exporter for system metrics. Prometheus supports floating point data, but not string data, so it is appropriate for system metrics but not logs.
- Use a dashboard tool such as **Kibana** or **Grafana** to visualize and monitor the data. The dashboard service can also run inside a container in the cluster.

For application logs, consider using **Fluentd** and **Elasticsearch**. Fluentd is an open source data collector, and Elasticsearch is a document database that is optimized to act as a search engine. Using this approach, each service sends logs to `stdout` and `stderr`, and Kubernetes writes these streams to the local file system. Fluentd collects the logs, optionally enriches them with additional metadata from Kubernetes, and sends the logs to Elasticsearch. Use Kibana, Grafana, or a similar tool to create a dashboard for Elasticsearch. Fluentd runs as a daemonset in the cluster, which ensures that one Fluentd pod is assigned to each node. You can configure Fluentd to collect kubelet logs as well as container logs. At high volumes, writing logs to the local file system could become a performance bottleneck, especially when multiple services are running on the same node. Monitor disk latency and file system utilization in production.

One advantage of using Fluentd with Elasticsearch for logs is that services do not require any additional library dependencies. Each service just writes to `stdout` and `stderr`, and Fluentd handles exporting the logs into Elasticsearch. Also, the teams writing services don't need to understand how to configure the logging infrastructure. One challenge is to configure the Elasticsearch cluster for a production deployment, so that it scales to handle your traffic.

Another option is to send logs to Operations Management Suite (OMS) Log Analytics. The Log Analytics service collects log data into a central repository, and can also consolidate data from other Azure services that your application uses. For more information, see Monitor an Azure Container Service cluster with Microsoft Operations Management Suite (OMS).

Example: Logging with correlation IDs

To illustrate some of the points discussed in this chapter, here is an extended example of how the Package service implements logging. The Package service was written in TypeScript and uses the Koa web framework for Node.js. There are several Node.js logging libraries to choose from. We picked Winston, a popular logging library that met our performance requirements when we tested it.

To encapsulate the implementation details, we defined an abstract `ILogger` interface:

```
export interface ILogger {
    log(level: string, msg: string, meta?: any)
    debug(msg: string, meta?: any)
    info(msg: string, meta?: any)
    warn(msg: string, meta?: any)
    error(msg: string, meta?: any)
}
```

Here is an `ILogger` implementation that wraps the Winston library. It takes the correlation ID as a constructor parameter, and injects the ID into every log message.

```
class WinstonLogger implements ILogger {
    constructor(private correlationId: string) {}
    log(level: string, msg: string, payload?: any) {
        var meta : any = {};
        if (payload) { meta.payload = payload };
        if (this.correlationId) { meta.correlationId = this.correlationId }
        winston.log(level, msg, meta)
    }

    info(msg: string, payload?: any) {
        this.log('info', msg, payload);
    }
    debug(msg: string, payload?: any) {
        this.log('debug', msg, payload);
    }
    warn(msg: string, payload?: any) {
        this.log('warn', msg, payload);
    }
    error(msg: string, payload?: any) {
        this.log('error', msg, payload);
    }
}
```

The Package service needs to extract the correlation ID from the HTTP request. For example, if you're using linkerd, the correlation ID is found in the `l5d-ctx-trace` header. In Koa, the HTTP request is stored in a Context object that gets passed through the request processing pipeline. We can define a middleware function to get the correlation ID from the Context and initialize the logger. (A middleware function in Koa is simply a function that gets executed for each request.)

```
export type CorrelationIdFn = (ctx: Context) => string;

export function logger(level: string, getCorrelationId: CorrelationIdFn) {
    winston.configure({
        level: level,
        transports: [new (winston.transports.Console)()]
        });
    return async function(ctx: any, next: any) {
        ctx.state.logger = new WinstonLogger(getCorrelationId(ctx));
        await next();
    }
}
```

This middleware invokes a caller-defined function, `getCorrelationId`, to get the correlation ID. Then it creates an instance of the logger and stashes it inside `ctx.state`, which is a key-value dictionary used in Koa to pass information through the pipeline.

The logger middleware is added to the pipeline on startup:

```
app.use(logger(Settings.logLevel(), function (ctx) {
    return ctx.headers[Settings.correlationHeader()];
}));
```

Once everything is configured, it's easy to add logging statements to the code. For example, here is the method that looks up a package. It makes two calls to the `ILogger.info` method.

```
async getById(ctx: any, next: any) {
  var logger : ILogger = ctx.state.logger;
  var packageId = ctx.params.packageId;
  logger.info('Entering getById, packageId = %s', packageId);

  await next();

  let pkg = await this.repository.findPackage(ctx.params.packageId)

  if (pkg == null) {
    logger.info(`getById: %s not found`, packageId);
    ctx.response.status= 404;
    return;
  }

  ctx.response.status = 200;
  ctx.response.body = this.mapPackageDbToApi(pkg);
}
```

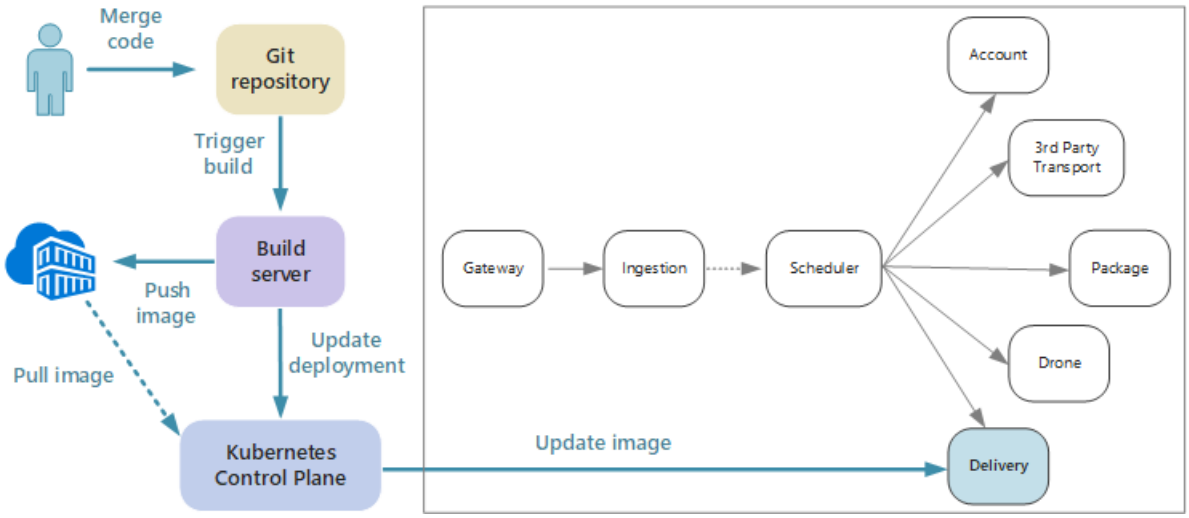
We don't need to include the correlation ID in the logging statements, because that's done automatically by the middleware function. This makes the logging code cleaner, and reduces the chance that a developer will forget to include the correlation ID. And because all of the logging statements use the abstract `ILogger` interface, it would be easy to replace the logger implementation later.

Continuous integration and delivery

Designing microservices: Continuous integration

12/12/2017 • 11 min to read • Edit Online

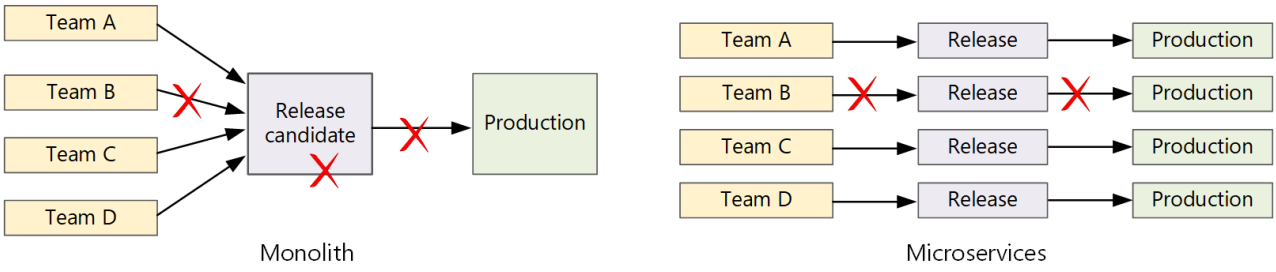
Continuous integration and continuous delivery (CI/CD) are a key requirement for achieving success with microservices. Without a good CI/CD process, you will not achieve the agility that microservices promise. Some of the CI/CD challenges for microservices arise from having multiple code bases and heterogenous build environments for the various services. This chapter describes the challenges and recommends some approaches to the problem.



Faster release cycles are one of the biggest reasons to adopt a microservices architecture.

In a purely monolithic application, there is a single build pipeline whose output is the application executable. All development work feeds into this pipeline. If a high-priority bug is found, a fix must be integrated, tested, and published, which can delay the release of new features. It's true that you can mitigate these problems by having well-factored modules and using feature branches to minimize the impact of code changes. But as the application grows more complex, and more features are added, the release process for a monolith tends to become more brittle and likely to break.

Following the microservices philosophy, there should never be a long release train where every team has to get in line. The team that builds service "A" can release an update at any time, without waiting for changes in service "B" to be merged, tested, and deployed. The CI/CD process is critical to making this possible. Your release pipeline must be automated and highly reliable, so that the risks of deploying updates are minimized. If you are releasing to production daily or multiple times a day, regressions or service disruptions must be very rare. At the same time, if a bad update does get deployed, you must have a reliable way to quickly roll back or roll forward to a previous version of a service.



When we talk about CI/CD, we are really talking about several related processes: Continuous integration, continuous delivery, and continuous deployment.

- Continuous integration means that code changes are frequently merged into the main branch, using automated build and test processes to ensure that code in the main branch is always production-quality.
- Continuous delivery means that code changes that pass the CI process are automatically published to a production-like environment. Deployment into the live production environment may require manual approval, but is otherwise automated. The goal is that your code should always be *ready* to deploy into production.
- Continuous deployment means that code changes that pass the CI/CD process are automatically deployed into production.

In the context of Kubernetes and microservices, the CI stage is concerned with building and testing container images, and pushing those images to a container registry. In the deployment stage, pod specs are updated to pick up the latest production image.

Challenges

- **Many small independent code bases**. Each team is responsible for building its own service, with its own build pipeline. In some organizations, teams may use separate code repositories. This could lead to a situation where the knowledge of how to build the system is spread across teams, and nobody in the organization knows how to deploy the entire application. For example, what happens in a disaster recovery scenario, if you need to quickly deploy to a new cluster?
- **Multiple languages and frameworks**. With each team using its own mix of technologies, it can be difficult to create a single build process that works across the organization. The build process must be flexible enough that every team can adapt it for their choice of language or framework.
- **Integration and load testing**. With teams releasing updates at their own pace, it can be challenging to design robust end-to-end testing, especially when services have dependencies on other services. Moreover, running a full production cluster can be expensive, so it's unlikely that every team will be able to run its own full cluster at production scales, just for testing.
- **Release management**. Every team should have the ability to deploy an update to production. That doesn't mean that every team member has permissions to do so. But having a centralized Release Manager role can reduce the velocity of deployments. The more that your CI/CD process is automated and reliable, the less there should be a need for a central authority. That said, you might have different policies for releasing major feature updates versus minor bug fixes. Being decentralized does not mean there should be zero governance.
- **Container image versioning**. During the development and test cycle, the CI/CD process will build many container images. Only some of those are candidates for release, and then only some of those release candidates will get pushed into production. You should have a clear versioning strategy, so that you know which images are currently deployed to production, and can roll back to a previous version if necessary.
- **Service updates**. When you update a service to a new version, it shouldn't break other services that depend on it. If you do a rolling update, there will be a period of time when a mix of versions is running.

These challenges reflect a fundamental tension. On the one hand, teams need to work as independently as possible. On the other hand, some coordination is needed so that a single person can do tasks like running an integration test, redeploying the entire solution to a new cluster, or rolling back a bad update.

CI/CD approaches for microservices

It's a good practice for every service team to containerize their build environment. This container should have all of the build tools necessary to build the code artifacts for their service. Often you can find an official Docker image for your language and framework. Then you can use `docker run` or Docker Compose to run the build.

With this approach, it's trivial to set up a new build environment. A developer who wants to build your code doesn't need to install a set of build tools, but simply runs the container image. Perhaps more importantly, your build server can be configured to do the same thing. That way, you don't need to install those tools onto the build server, or manage conflicting versions of tools.

For local development and testing, use Docker to run the service inside a container. As part of this process, you may need to run other containers that have mock services or test databases needed for local testing. You could use Docker Compose to coordinate these containers, or use Minikube to run Kubernetes locally.

When the code is ready, open a pull request and merge into master. This will start a job on the build server:

1. Build the code assets.
2. Run unit tests against the code.
3. Build the container image.
4. Test the container image by running functional tests on a running container. This step can catch errors in the Docker file, such as a bad entry point.
5. Push the image to a container registry.
6. Update the test cluster with the new image to run integration tests.

When the image is ready to go into production, update the deployment files as needed to specify the latest image, including any Kubernetes configuration files. Then apply the update to the production cluster.

Here are some recommendations for making deployments more reliable:

- Define organization-wide conventions for container tags, versioning, and naming conventions for resources deployed to the cluster (pods, services, and so on). That can make it easier to diagnose deployment issues.
- Create two separate container registries, one for development/testing and one for production. Don't push an image to the production registry until you're ready to deploy it into production. If you combine this practice with semantic versioning of container images, it can reduce the chance of accidentally deploying a version that wasn't approved for release.

Updating services

There are various strategies for updating a service that's already in production. Here we discuss three common options: Rolling update, blue-green deployment, and canary release.

Rolling update

In a rolling update, you deploy new instances of a service, and the new instances start receiving requests right away. As the new instances come up, the previous instances are removed.

Rolling updates are the default behavior in Kubernetes when you update the pod spec for a Deployment. The Deployment controller creates a new ReplicaSet for the updated pods. Then it scales up the new ReplicaSet while scaling down the old one, to maintain the desired replica count. It doesn't delete old pods until the new ones are ready. Kubernetes keeps a history of the update, so you can use kubectl to roll back an update if needed.

If your service performs a long startup task, you can define a readiness probe. The readiness probe reports when the container is ready to start receiving traffic. Kubernetes won't send traffic to the pod until the probe reports success.

One challenge of rolling updates is that during the update process, a mix of old and new versions are running and receiving traffic. During this period, any request could get routed to either of the two versions. That may or may not cause problems, depending on the scope of the changes between the two versions.

Blue-green deployment

In a blue-green deployment, you deploy the new version alongside the previous version. After you validate the

new version, you switch all traffic at once from the previous version to the new version. After the switch, you monitor the application for any problems. If something goes wrong, you can swap back to the old version. Assuming there are no problems, you can delete the old version.

With a more traditional monolithic or N-tier application, blue-green deployment generally meant provisioning two identical environments. You would deploy the new version to a staging environment, then redirect client traffic to the staging environment — for example, by swapping VIP addresses.

In Kubernetes, you don't need to provision a separate cluster to do blue-green deployments. Instead, you can take advantage of selectors. Create a new Deployment resource with a new pod spec and a different set of labels. Create this deployment, without deleting the previous deployment or modifying the service that points to it. Once the new pods are running, you can update the service's selector to match the new deployment.

An advantage of blue-green deployments is that the service switches all the pods at the same time. After the service is updated, all new requests get routed to the new version. One drawback is that during the update, you are running twice as many pods for the service (current and next). If the pods require a lot of CPU or memory resources, you may need to scale out the cluster temporarily to handle the resource consumption.

Canary release

In a canary release, you roll out an updated version to a small number of clients. Then you monitor the behavior of the new service before rolling it out to all clients. This lets you do a slow rollout in a controlled fashion, observe real data, and spot problems before all customers are affected.

A canary release is more complex to manage than either blue-green or rolling update, because you must dynamically route requests to different versions of the service. In Kubernetes, you can configure a Service to span two replica sets (one for each version) and adjust the replica counts manually. However, this approach is rather coarse-grained, because of the way Kubernetes load balances across pods. For example, if you have a total of ten replicas, you can only shift traffic in 10% increments. If you are using a service mesh, you can use the service mesh routing rules to implement a more sophisticated canary release strategy. Here are some resources that may be helpful:

- Kubernetes without service mesh: Canary deployments
- Linkerd: Dynamic request routing
- Istio: Canary Deployments using Istio

Conclusion

In recent years, there has been a sea change in the industry, a movement from building *systems of record* to building *systems of engagement*.

Systems of record are traditional back-office data management applications. At the heart of these systems there often sits an RDBMS that is the single source of truth. The term "system of engagement" is credited to Geoffrey Moore, in his 2011 paper *Systems of Engagement and the Future of Enterprise IT*. Systems of engagement are applications focused on communication and collaboration. They connect people in real time. They must be available 24/7. New features are introduced regularly without taking the application offline. Users expect more and are less patient of unexpected delays or downtime.

In the consumer space, a better user experience can have measurable business value. The amount of time that a user engages with an application may translate directly into revenue. And in the realm of business systems, users' expectations have changed. If these systems aim to foster communication and collaboration, they must take their cue from consumer-facing applications.

Microservices are a response to this changing landscape. By decomposing a monolithic application into a group of loosely coupled services, we can control the release cycle of each service, and enable frequent updates without downtime or breaking changes. Microservices also help with scalability, failure isolation, and resiliency. Meanwhile, cloud platforms are making it easier to build and run microservices, with automated provisioning of compute

resources, container orchestrators as a service, and event-driven serverless environments.

But as we've seen, microservices architectures also being a lot of challenges. To succeed, you must start from a solid design. You must put careful thought into analyzing the domain, choosing technologies, modeling data, designing APIs, and building a mature DevOps culture. We hope that this guide, and the accompanying reference implementation, has helped to illuminate the journey.