IT457 – Cloud Computing **Lab Report – 7**

Date – April 23, 2023

Name – Divya Kirtikumar Patel

Student ID - 202001420

Given Scenario

Fabrikam, Inc. will deliver via drone. It operates drones. Businesses can register and request a drone to deliver items. A backend system allocates a drone and provides an expected delivery time when a customer orders a pickup. The customer can track the drone and its ETA during delivery.

This is perfect for aerospace and aviation.

This domain is complex. Schedule drones, track packages, manage user accounts, and store and analyse previous data. Fabrikam also wants to launch rapidly and iterate with new features. The cloud-scale application must have a high service level target (SLO). Fabrikam expects data storage and querying needs to vary by system part. Fabrikam chose microservices architecture for these reasons.

Choosing an Azure compute option for microservices

Service orchestrators

- Orchestrators deploy and manage services. These include installing services on nodes, monitoring service health, restarting unhealthy services, load balancing network traffic across service instances, service discovery, scaling service instances, and applying configuration modifications. Kubernetes, Service Fabric, DC/OS, and Docker Swarm are popular orchestrators.
- Options on Azure:
- AKS manages Kubernetes. AKS hosts and manages the Kubernetes control plane, automating updates, patching, autoscaling, and other administration activities. "Kubernetes APIs as a service" is AKS.
- Kubernetes-based Azure Container Apps simplifies container orchestration and other management duties. Container Apps facilitates serverless containerized application and microservice deployment and management with Kubernetes features.
- Distributed systems platform Service Fabric packages, deploys, and manages microservices. Service Fabric deploys microservices as containers, binary executables, or Reliable Services. Reliable Services services can directly use Service Fabric programming APIs to query the system, report health, receive notifications about configuration and code changes, and identify other services. Service Fabric emphasises stateful services utilising Reliable Collections.
- Azure IaaS supports Docker Enterprise Edition and other choices. Azure Marketplace has deployment templates.

Containers

- Containers and microservices are sometimes conflated. While it's not true—you don't require containers to construct microservices—containers provide benefits that are particularly relevant to microservices, such as:
- Portability. A container image runs without dependencies or libraries. That simplifies deployment. To manage increased demand or recover from node failures, start and stop containers rapidly.
- Density. Sharing OS resources makes containers lighter than virtual machines. When the application has several little services, that lets you pack multiple containers onto a node.
- Resource isolation. Limiting a container's memory and CPU can prevent a runaway process from exhausting host resources. See Bulkhead pattern.

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.

Design interservice communication for microservices

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

Challenges

Here are some of the main challenges arising from service-to-service communication. Service meshes, described later in this article, 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 article 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.

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 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.

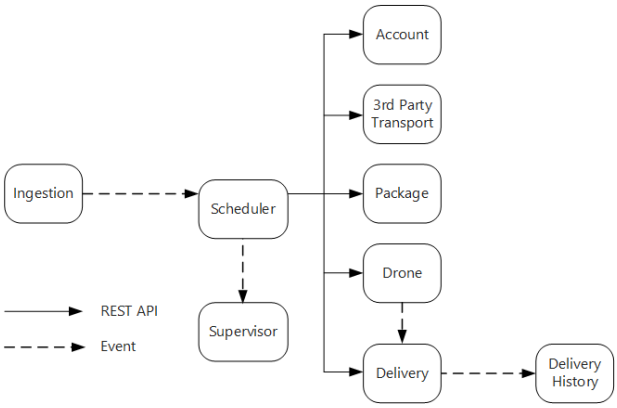
Drone Delivery: Choosing the messaging patterns

This solution uses the Drone Delivery example. It's ideal for the aerospace and aircraft industries.

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

- Client applications plan, change, and cancel deliveries using the Ingestion service's REST API.
- Event Hubs send Scheduler service asynchronous messages from the Ingestion service. Ingestion load-leveling requires asynchronous messaging.
- Account, Delivery, Package, Drone, and Third-party Transport have internal REST APIs. These APIs fulfil user requests from the Scheduler service. Synchronous APIs are used because the Scheduler needs responses from downstream services. Any downstream service failure ends the process. Calling backend services may cause latency.
- If a downstream service fails nontransiently, the transaction should fail. The Supervisor schedules compensating transactions once the Scheduler service sends an asynchronous message.
- Clients can use the Delivery service API to check delivery status. An API gateway can hide the underlying services from the client, so the client doesn't need to know which services provide which APIs.
- The Drone service delivers drone location and status events while in flight. The Delivery service tracks delivery status by listening to these events.

- Delivery status events like DeliveryCreated and DeliveryCompleted are sent by the Delivery service. Any service can subscribe. The Delivery History service is the only subscriber now, but more may join later. Real-time analytics services may receive events. Adding subscribers doesn't alter the main workflow path because the Scheduler doesn't wait for a response.



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. Currently, the service mesh concept applies mainly to container orchestrators, rather than serverless architectures.

Distributed transactions

A common challenge in microservices is correctly handling transactions that span multiple services. Often in this scenario, the success of a transaction is all or nothing — if one of the participating services fails, the entire transaction must fail.

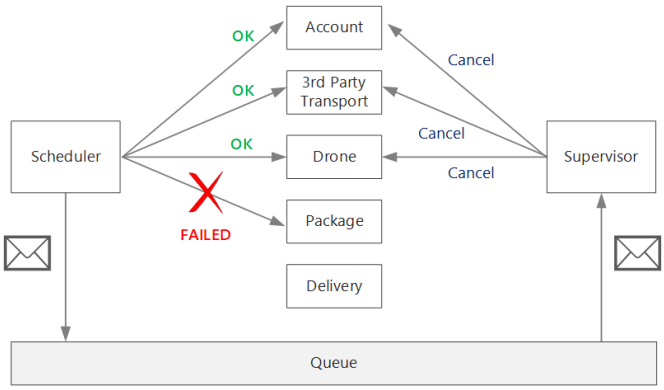
There are two cases to consider:

- A 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 nontransient failure.
- A *nontransient* failure is any failure that's unlikely to go away by itself. Nontransient failures include normal error conditions, such as invalid input. 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.

After a nontransient failure, 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.



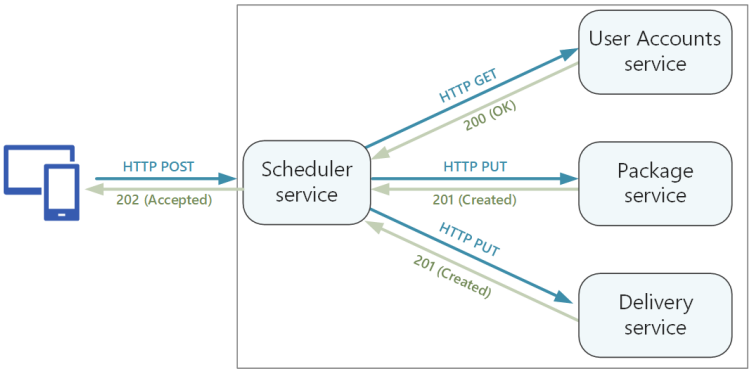
The Scheduler service itself might fail (for example, because a node crashes). In that case, a new instance can spin up and take over. However, any transactions that were already in progress must be resumed.

One approach is to save a checkpoint to a durable store after each step in the workflow is completed. If an instance of the Scheduler service crashes in the middle of a transaction, a new instance can use the checkpoint to resume where the previous instance left off. However, writing checkpoints can create a performance overhead.

Another option 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. Essentially, the downstream service should ignore duplicate calls, which means the service must be able to detect duplicate calls. It's not always straightforward to implement idempotent methods. For more information, see Idempotent operations.

Design APIs for microservices

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.



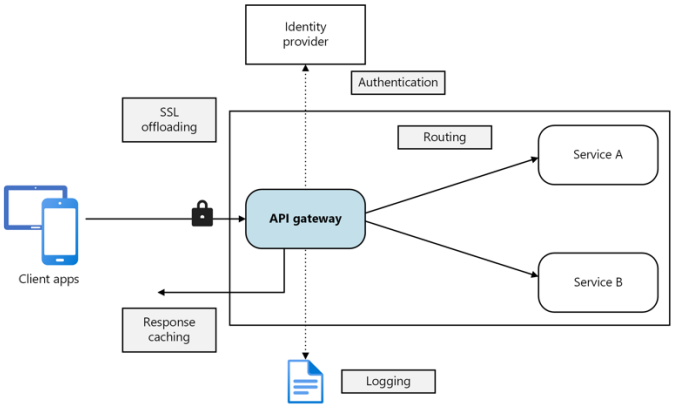
It's important to distinguish between two types of API:

- 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.

Use API gateways in microservices

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.



Microsoft Azure

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 require 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 allowlist or blocklist
- 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 referred to as NGINX JavaScript. This module was formally named 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 Front Door is Microsoft's modern cloud Content Delivery Network (CDN) that provides fast, reliable, and secure access between your users and your applications' static and dynamic web content across the globe. Azure Front Door delivers your content using the Microsoft's global edge network with hundreds of global and local points of presence (PoPs) distributed around the world close to both your enterprise and consumer end users.
- 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 restrictions, 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. For information about using API Management with Application Gateway, see Integrate API Management in an internal VNet with Application Gateway.

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.

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.

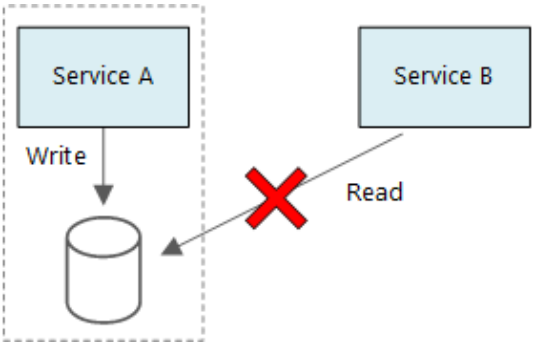
- Stable configuration. If the gateway is misconfigured, the entire application may become unavailable.
- Performance. You may want to use a specific VM configuration for the gateway for performance reasons.
-

Data considerations for microservices

This article 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.

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.

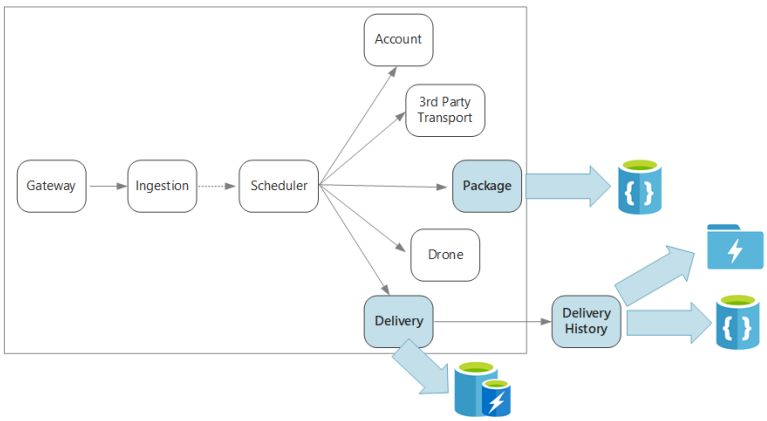
- For transactions, use patterns such as Scheduler Agent Supervisor and Compensating Transaction to keep data consistent across several services. You may need to store an additional piece of data that captures the state of a unit of work that spans multiple services, to avoid partial failure among multiple 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.

Choosing data stores for the Drone Delivery application

The previous articles in this series discuss a drone delivery service as a running example. You can read more about the scenario and the corresponding reference implementation here. This example is ideal for the aircraft and aerospace industries.

To recap, this application defines several microservices for scheduling deliveries by drone. When a user schedules a new delivery, the client request includes information about the delivery, such as pickup and dropoff locations, and about the package, such as size and weight. This information defines a unit of work.

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 Cache for Redis 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 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 Azure Cosmos DB for quicker lookup. The records don't need to stay in Azure Cosmos DB indefinitely. Older deliveries can be archived — say, after a month. This could be done by running an occasional batch process. Archiving older data can reduce costs for Cosmos DB while still keeping the data available for historical reporting from the Data Lake.

Package service

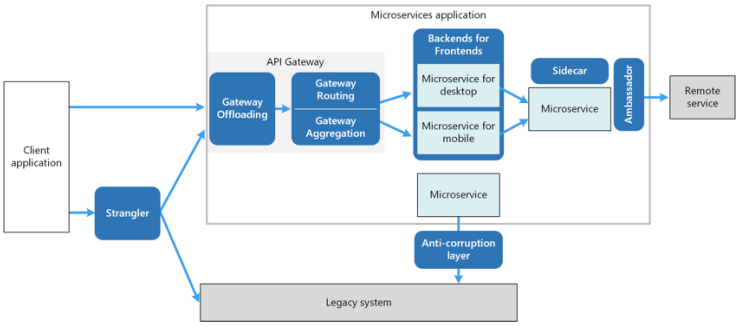
The Package service stores information about all of the packages. The storage requirements for the Package are:

- 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 Azure Cosmos DB can achieve 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 Azure Cosmos DB. That lets them leverage their existing experience with MongoDB, while getting the benefits of Azure Cosmos DB, which is a managed Azure service.

Design patterns for microservices

The goal of microservices is to increase the velocity of application releases, by decomposing the application into small autonomous services that can be deployed independently. A microservices architecture also brings some challenges. The design patterns shown here can help mitigate these challenges.



Ambassador can be used to offload common client connectivity tasks such as monitoring, logging, routing, and security (such as TLS) in a language agnostic way. Ambassador services are often deployed as a sidecar (see below).

Anti-corruption layer implements a façade between new and legacy applications, to ensure that the design of a new application is not limited by dependencies on legacy systems.

Backends for Frontends creates separate backend services for different types of clients, such as desktop and mobile. That way, a single backend service doesn't need to handle the conflicting requirements of various client types. This pattern can help keep each microservice simple, by separating client-specific concerns.

Bulkhead isolates critical resources, such as connection pool, memory, and CPU, for each workload or service. By using bulkheads, a single workload (or service) can't consume all of the resources, starving others. This pattern increases the resiliency of the system by preventing cascading failures caused by one service.

Gateway Aggregation aggregates requests to multiple individual microservices into a single request, reducing chattiness between consumers and services.

Gateway Offloading enables each microservice to offload shared service functionality, such as the use of SSL certificates, to an API gateway.

Gateway Routing routes requests to multiple microservices using a single endpoint, so that consumers don't need to manage many separate endpoints.

Sidecar deploys helper components of an application as a separate container or process to provide isolation and encapsulation.

Strangler Fig supports incremental refactoring of an application, by gradually replacing specific pieces of functionality with new services.