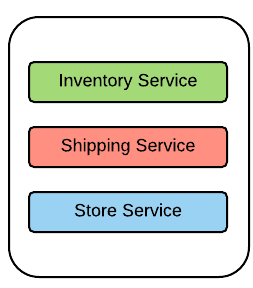
# Microservices in Practice: From Architecture to Deployment

## Monolithic Architecture

Enterprise software applications are designed to facilitate numerous business requirements; a given software application offers hundreds of functionalities and all such functionalities are piled into a single monolithic application. For example, ERPs, CRMs, and other various software systems are built as a monolith with several hundreds of functionalities. The deployment, troubleshooting, scaling, and upgrading of such monstrous software applications is a nightmare.

Service Oriented Architecture (SOA) was designed to overcome some of the aforementioned limitations by introducing the concept of a service, an aggregation and grouping of similar functionalities offered from an application. With SOA, a software application is designed as a combination of coarse-grained services. However, in SOA, the scope of a service is very broad. That leads to complex and mammoth services with several dozens of operations (functionalities), along with complex message formats and standards (e.g: all WS\* standards).



Monolithic architecture

In most cases, services in SOA are independent of each other; yet they are deployed in the same runtime along with all other services (just think about having several web applications that are deployed into the same Tomcat instance). Similar to monolithic software applications, these services have a habit of growing over time by accumulating various functionalities. Literally, that turns those applications into monolithic globs that are no different from conventional monolithic applications such as ERPs. The shows a retail software application which comprises of multiple services. All these services are deployed into the same application runtime. So, it's a very good example of a monolithic architecture. Here are some of the characteristics of applications based on monolithic architecture.

* Monolithic applications are designed, developed, and deployed as a single unit.
* Monolithic applications are overwhelmingly complex; this leads to nightmares in maintaining, upgrading, and adding new features.
* It is difficult to practice agile development and delivery methodologies with Monolithic architecture.
* It is required to redeploy the entire application in order to update a part of it.
* The application has to be scaled as a single unit, making it difficult to manage conflicting resource requirements (e.g. one service requires more CPU, while the other requires more memory)
* One unstable service can bring the whole application down.
* It's really difficult to adopt new technologies and frameworks, as of all the functionalities have to build on homogeneous technologies/frameworks.

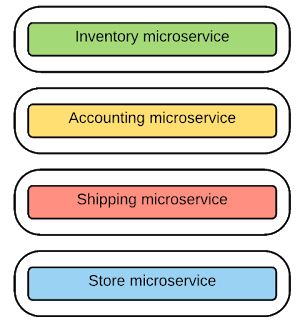
## Microservices Architecture

The foundation of Micorservices architecture (MSA) is about developing a single application as a suite of small and independent services that are run in their own process, developed, and deployed independently.

In most of the definitions of Micorservices architecture, it is explained as the process of segregating the services available in the monolith into a set of independent services. However, in my opinion, Microservices is not just about splitting the services available in monolith into independent services.

The key idea is that by looking at the functionalities offered from the monolith, we can identify the required business capabilities. Then those business capabilities can be implemented as fully independent, fine-grained, and self-contained (micro)services. They might be implemented on top of different technology stacks and each service is addressing a very specific and limited business scope.

Therefore, the online retail system scenario that we explain above can be realized with Micorservices architecture, as depicted in the figure below. With microservice architecture, the retail software application is implemented as a suite of Micorservices. So, as you can see below, based on business requirements, there is an additional microservice created from the original set of services that are there in the monolith. So, it is quite obvious that using microservices architecture is something beyond the splitting of the services in the monolith.



Microservices architecture

Let's dive deep into the key architectural principles of Micorservices, and more importantly, let's focus on how they can be used in practice.

### Designing Microservices: Size, Scope, and Capabilities

You may be building your software application from scratch by using Microservices Architecture, or you may be converting existing applications/services into Micorservices. Either way, it is quite important that you properly decide the size, scope, and capabilities of the Microservices. Probably, that is the hardest thing that you initially encounter when you implement Microservices Architecture in practice.

Let's discuss some of the key practical concerns and misconceptions related to the size, scope, and capabilities of microservices.

* **Lines of Code/Team size are lousy metrics**: There are several discussions on deciding the size of the Microservices based on the lines-of-code of its implementation or its team's size (i.e. [two-pizza team](http://blog.idonethis.com/two-pizza-team/)). However, these are considered to be very impractical and lousy metrics, because we can still develop services with less code/with two-pizza-team size but totally violating the microservice architectural principals.
* **"Micro" is a bit misleading term**: Most developers tend to think that they should try to make the service as small as possible. This is a misconception.
* **SOA context**: In the SOA context, services are often implemented as monolithic globs with the support for several dozens of operations/functionalities. So, having SOA-like services and rebranding them as microservices is not going to give you any benefits of microservices architecture.

So, then how should we properly design services in Microservices Architecture?

#### Guidelines for Designing Microservices

* Single Responsibility Principle(SRP): Having a limited and focused business scope for a microservice helps us to meet agility in development and delivery of services.
* During the designing phase of the microservices, we should find their boundaries and align them with business capabilities (also known as bounded context in [Domain-Driven-Design](http://martinfowler.com/bliki/BoundedContext.html)).
* Make sure the microservices design ensures the agile/independent development and deployment of the service.
* Our focus should be on the scope of the microservice but not about making the service smaller. The (right) size of the service should be the required size to facilitate a given business capability.
* Unlike service in SOA, a given microservice should have very few operations/functionalities and a simple message format.
* It is often a good practice to start with relatively broad service boundaries to begin with and then refactor to smaller ones (based on business requirements) as time goes on.

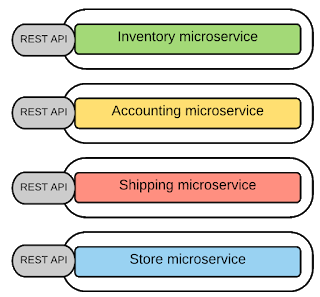
In our retail use case, you can find that we have split the functionalities of the monolith into four different microservices, namely "inventory," "accounting," "shipping," and "store." They are addressing a limited but focused business scope so that each service is fully decoupled from each other and ensures the agility in development and deployment.

### Messaging in Microservices

In monolithic applications, business functionalities of different processors/components are invoked using function calls or language-level method calls. In SOA, this was shifted towards a much more loosely coupled web service level messaging, which is primarily based on SOAP on top of different protocols such as HTTP, JMS. Webservices with several dozens of operations and complex message schemas was a key resistive force for the popularity of web services. For Microservices architecture, it is required to have a simple and lightweight messaging mechanism.

#### Synchronous Messaging - REST, Thrift

For synchronous messaging (client expects a timely response from the service and waits till it get it) in Microservices Architecture, [REST](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm) is the unanimous choice as it provides a simple messaging style implemented with HTTP request-response, based on resource API style. Therefore, most microservices implementations are using HTTP along with resource API based styles (every functionality is represented with a resource and operations carried out on top of those resources).



Using REST interfaces to expose microservices

[Thrift](https://thrift.apache.org/) is used (in which you can define an interface definition for your microservice), as an alternative to REST/HTTP synchronous messaging.

#### Asynchronous Messaging - AMQP, STOMP, MQTT

For some microservices scenarios, it is required to use asynchronous messaging techniques (client doesn't expect a response immediately, or does not accept a response at all). In such scenarios, asynchronous messaging protocols such as [AMQP](https://www.amqp.org/), [STOMP](http://stomp.github.io/), or [MQTT](http://mqtt.org/) are widely used.

#### Message Formats - JSON, XML, Thrift, ProtoBuf, Avro

Deciding the best message format for Microservices is another key factor. The traditional monolithic applications use complex binary formats, SOA/Web services-based applications use text messages based on complex message formats (SOAP) and schemas (xsd). In most microservices-based applications, they use simple text-based message formats, such as JSON and XML on top of HTTP resource API style. In cases where we need binary message formats (text messages can become verbose in some use cases), microservices can leverage binary message formats such as binary Thrift, [ProtoBuf](https://github.com/google/protobuf), or [Avro](https://avro.apache.org/).

#### Service Contracts - Defining the Service Interfaces - Swagger, RAML, Thrift IDL

When you have a business capability implemented as a service, you need to define and publish the service contract. In traditional monolithic applications, we barely find such features to define the business capabilities of an application. In SOA/Web services world, WSDL is used to define the service contract, but, as we all know, WSDL is not the ideal solution for defining microservices contract as WSDL is insanely complex and tightly coupled to SOAP.

Since we build microservices on top of a REST architectural style, we can use the same REST API definition techniques to define the contract of microservices. Therefore, microservices use the standard REST API definition languages, such as [Swagger](http://swagger.io/) and [RAML](http://raml.org/) to define the service contracts.

For other microservices implementation that are not based on HTTP/REST (such as Thrift), we can use the protocol level Interface Definition Languages(IDL) (e.g.: [Thrift IDL](https://thrift.apache.org/docs/idl)).

### Integrating Microservices (Inter-service/process Communication)

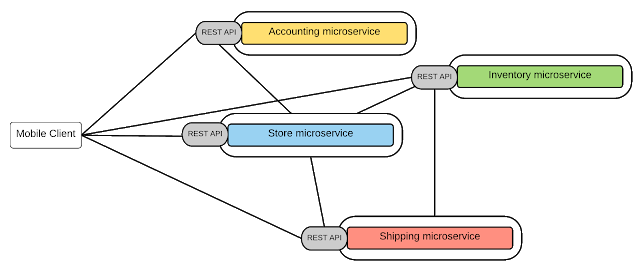
In Microservices architecture, the software applications are built as a suite of independent services. So, in order to realize a business use case, it is required to have the communication structures between different microservices/processes. That's why inter-service/process communication between microservices is such a vital aspect.

In SOA implementations, the inter-service communication between services is facilitated with an Enterprise Service Bus (ESB) and most of the business logic resides in the intermediate layer (message routing, transformation, and orchestration). However, Microservices architecture promotes to eliminate the central message bus/ESB and move the "smart-ness" or business logic to the services and client (known as Smart Endpoints).

Since microservices use standard protocols such as HTTP, JSON, etc. the requirement of integrating with a disparate protocol is minimal when it comes to the communication among microservices. Another alternative approach in Microservice communication is to use a lightweight message bus or [gateway](http://martinfowler.com/eaaCatalog/gateway.html) with minimal routing capabilities and act as a "dumb pipe" with no business logic implemented on the gateway. Based on these styles, there are several communication patterns that have emerged in microservices architecture.

#### Point-to-point Style - Invoking Services Directly

In point to point style, the entirety of the message routing logic resides on each endpoint and the services can communicate directly. Each microservice exposes a REST APIs and a given microservice or an external client can invoke another microservice through its REST API.



Inter-service communication with point-to-point connectivity

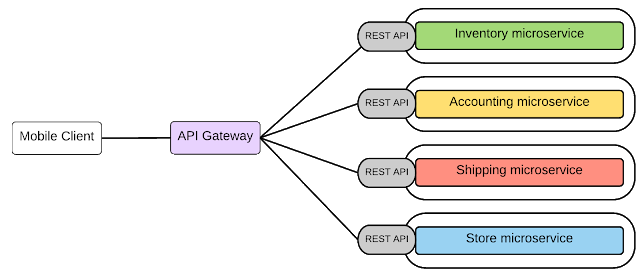
Obviously, this model works for relatively simple microservices-based applications, but as the number of services increases, this will become overwhelmingly complex. After all, that's the exact reason for using ESB in the traditional SOA implementation: to get rid of the messy point-to-point integration links. Let's try to summarize the key drawbacks of the point-to-point style for microservice communication.

* The non-functional requirements such as end-user authentication, throttling, monitoring, etc. has to be implemented at each and every microservice level.
* As a result of duplicating common functionalities, each microservice implementation can become complex.
* There is no control at all of the communication between the services and clients (even for monitoring, tracing, or filtering)
* Often the direct communication style is considered as a microservice [anti-pattern](http://www.infoq.com/articles/seven-uservices-antipatterns) for large scale microservice implementations.

Therefore, for complex Microservices use cases, rather than having point-to-point connectivity or a central ESB, we could have a lightweight central messaging bus which can provide an abstraction layer for the microservices and that can be used to implement various non-functional capabilities. This style is known as API Gateway style.

#### API-Gateway Style

The key idea behind the API Gateway style is that using a lightweight message gateway as the main entry point for all the clients/consumers and implement the common non-functional requirements at the Gateway level. In general, an API Gateway allows you to consume a managed API over REST/HTTP. Therefore, here we can expose our business functionalities which are implemented as microservices, through the API-GW, as managed APIs. In fact, this is a combination of Microservices architecture and API-Management which give you the best of both worlds.



All microservices are exposed through an API-GWIn our retail business scenario, as depicted in figure above, all the microservices are exposed through an API-GW and that is the single entry point for all the clients. If a microservice wants to consume another microservice that also needs to be done through the API-GW.

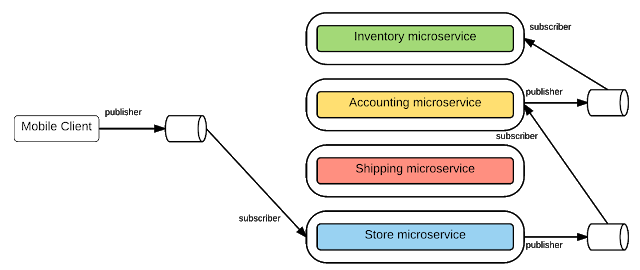
API-GW style gives you the following advantages:

* Ability to provide the required abstractions at the gateway level for the existing microservices. For example, rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client.
* Lightweight message routing/transformations at gateway level.
* Central place to apply non-functional capabilities such as security, monitoring and throttling.
* With the use of API-GW pattern, the microservice becomes even more lightweight as all the non-functional requirements are implemented at the Gateway level.

The API-GW style could well be the most widely used pattern in most microservice [implementations](http://techblog.netflix.com/2012/07/embracing-differences-inside-netflix.html).

#### Message Broker style

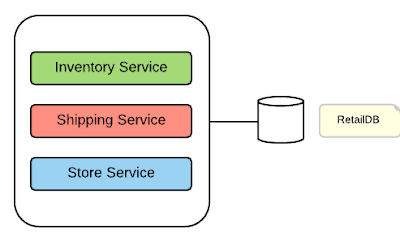
The microservices can be integrated with an asynchronous messaging scenario such as one-way requests and publish-subscribe messaging using queues or topics. A given microservice can be the message producer and it can asynchronously send messages to a queue or topic. Then the consuming microservice can consume messages from the queue or topic. This style decouples message producers from message consumers and the intermediate message broker will buffer messages until the consumer is able to process them. Producer microservices are completely unaware of the consumer microservices.

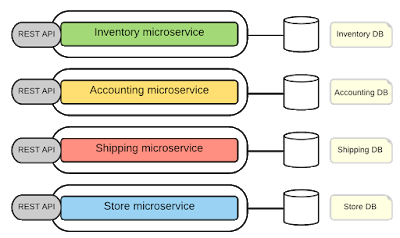
Asynchronous messaging-based integration using pub-sub

The communication between the consumers/producers is facilitated through a message broker which is based on asynchronous messaging standards such as AMQP, MQTT, etc.

### Decentralized Data Management

In a monolithic architecture, the application stores data in a single and centralized databases to implement various functionalities/capabilities of the application.

Monolithic application using a centralized database to implement all its features.In Microservices architecture, the functionalities are dispersed across multiple microservices and, if we use the same centralized database, then the microservices will no longer be independent from each other (for instance, if the database schema has changed from a given microservice, that will break several other services). Therefore, each microservice has to have its own database.

Each microservices has its own private database

Here are the key aspects of implementing decentralized data management in microservices architecture.

* Each microservice can have a private database to persist the data that requires to implement the business functionality offered from it.
* A given microservice can only access the dedicated private database but not the databases of other microservices.
* In some business scenarios, you might have to update several database for a single transaction. In such scenarios, the databases of other microservices should be updated through its service API only (not allowed to access the database directly)

The decentralized data management gives you the fully decoupled microservices and the liberty of choosing disparate data management techniques (SQL or NoSQL etc., different database management systems for each service). However, for complex transactional use cases that involve multiple microservices, the transactional behavior has to be implemented using the APIs offered from each service and the logic resides either at the client or intermediary (GW) level.

### Decentralized Governance

Microservices architecture favors decentralized governance.

In general, 'governance' means establishing and enforcing how people and solutions work together to achieve organizational objectives. In the context of SOA, [SOA governance](https://www.ibm.com/developerworks/webservices/library/ar-servgov/) guides the development of reusable services, establishing how services will be designed and developed and how those services will change over time. It establishes agreements between the providers of services and the consumers of those services, telling the consumers what they can expect and the providers what they're obligated to provide. In SOA Governance there are two types of governance that are in common use:

* Design-time governance - defining and controlling the service creations, design, and implementation of service policies
* Run-time governance - the ability to enforce service policies during execution

So, what does governance in a microservices context really mean? In microservices architecture, the microservices are built as fully independent and decoupled services with a variety of technologies and platforms. So, there is no need for defining a common standard for services design and development. So, we can summarize the decentralized governance capabilities of Microservices as follows:

* In microservices architecture, there is no requirement to have centralized design-time governance.
* Microservices can make their own decisions about its design and implementation.
* Microservices architecture fosters the sharing of common/reusable services.
* Some of the run-time governances aspects, such as SLAs, throttling, monitoring, common security requirements, and service discovery may be implemented at API-GW level.

### Service Registry and Service Discovery

In Microservices architecture, the number of microservices that you need to deal with is quite high. And also, their locations change dynamically owing to the rapid and agile development/deployment nature of microservices. Therefore, you need to find the location of a microservice during the runtime. The solution to this problem is to use a Service Registry.

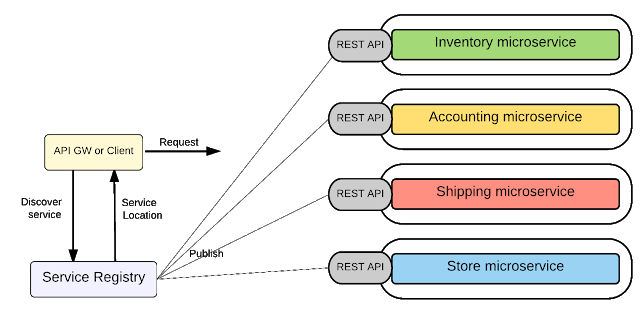
#### Service Registry

Service Registry holds the microservices instances and their locations. Microservice instances are registered with the service registry on startup and deregistered on shutdown. The consumers can find the available microservices and their locations through service registry.

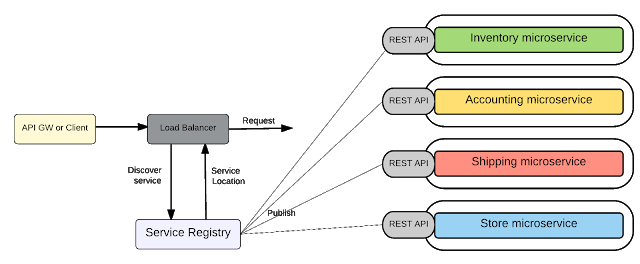
#### Service Discovery

To find the available microservices and their location, we need to have a service discovery mechanism. There are two types of service discovery mechanisms, Client-side Discovery and Server-side Discovery. Let's have a closer look at those service discovery mechanisms.

Client-side Discovery — In this approach, the client or the API-GW obtains the location of a service instance by querying a Service Registry.

Client-side discovery

Server-side Discovery — With this approach, clients/API-GW sends the request to a component (such as a Load balancer) that runs on a well-known location. That component calls the service registry and determines the absolute location of the microservice.

Client-side discovery

The microservices deployment solutions, such as Kubernetes (<http://kubernetes.io/v1.1/docs/user-guide/services.html>), offers service-side discovery mechanisms.

### Deployment

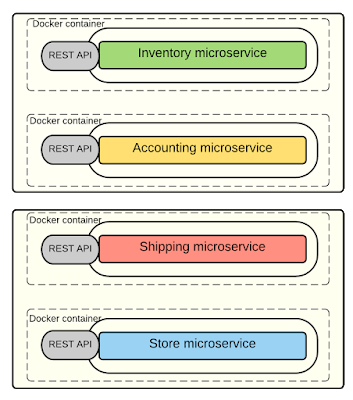
When it comes to microservices architecture, the deployment of microservices plays a critical role and has the following key requirements:

* Ability to deploy/un-deploy independently of other microservices.
* Must be able to scale at each microservices level (a given service may get more traffic than other services).
* Building and deploying microservices quickly.
* Failure in one microservice must not affect any of the other services.

[Docker](https://www.docker.com/) (an open source engine that lets developers and system administrators deploy self-sufficient application containers in Linux environments) provides a great way to deploy microservices addressing the above requirements. The key steps involved are as follows:

* Package the microservice as a (Docker) container image.
* Deploy each service instance as a container.
* Scaling is done based on changing the number of container instances.
* Building, deploying, and starting microservice will be much faster as we are using Docker containers (which is much faster than a regular VM)

[Kubernetes](http://kubernetes.io/) is extending Docker's capabilities by allowing to manage a cluster of Linux containers as a single system, managing and running Docker containers across multiple hosts, offering co-location of containers, service discovery, and replication control. As you can see, most of these features are essential in our microservices context too. Hence using Kubernetes (on top of Docker) for microservices deployment has become an extremely powerful approach, especially for large scale microservices deployments.

Building and deploying microservices as containers.

In the figure above, it shows an overview of the deployment of the microservices of the retail application. Each microservice instance is deployed as a container and there are two containers per each host. You can arbitrarily change the number of containers that you run on a given host.

### Security

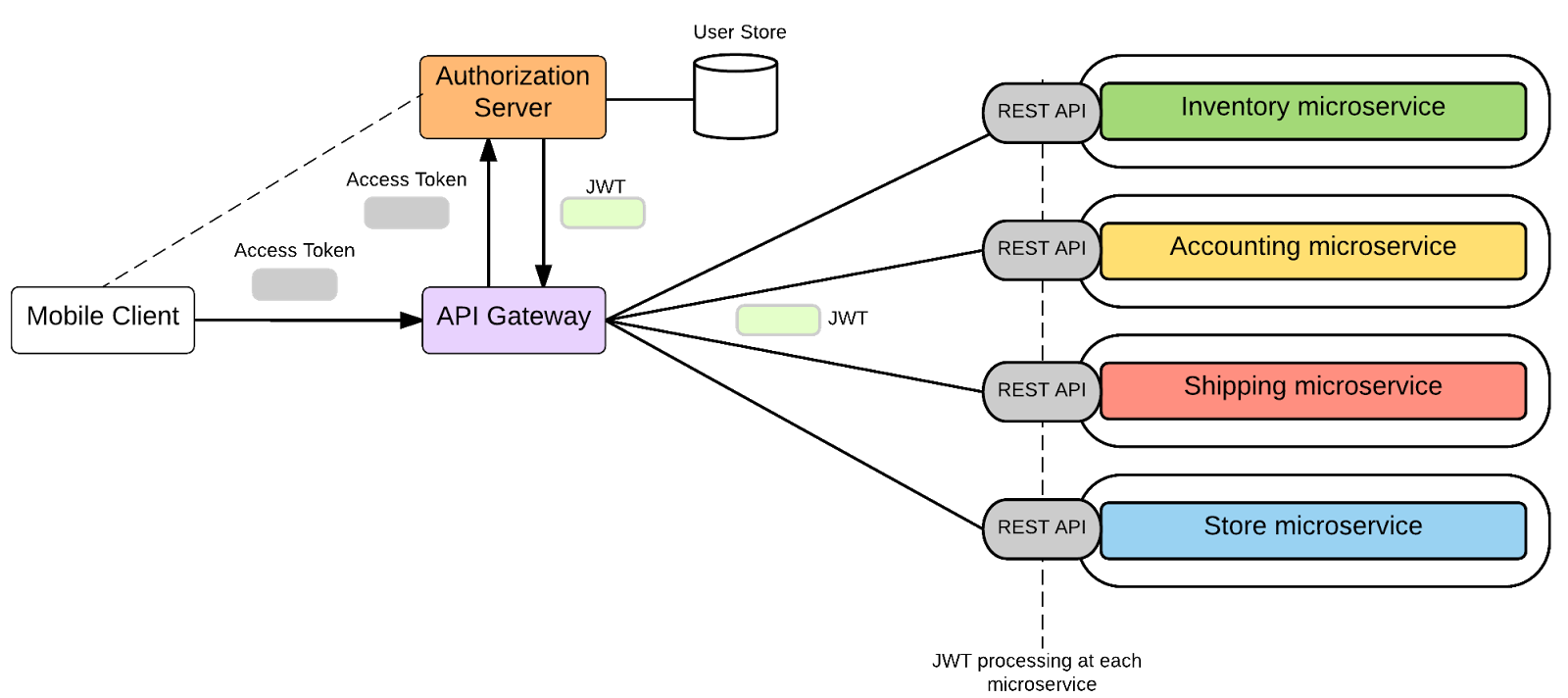
Securing microservices is a quite common requirement when you use microservices in real-world scenarios. Before jumping into microservices security, let's have a quick look at how we normally implement security at the monolithic application level.

* In a typical monolithic application, the security is about finding that 'who is the caller', 'what can the caller do', and 'how do we propagate that information'.
* This is usually implemented at a common security component which is at the beginning of the request handling chain and that component populates the required information with the use of an underlying user repository (or user store).

So, can we directly translate this pattern into the microservices architecture? Yes, but that requires a security component implemented at each microservices level which is talking to a centralized/shared user repository and retrieve the required information. That's is a very tedious approach of solving the Microservices security problem. Instead, we can leverage the widely used API-Security standards such as OAuth2 and OpenID Connect to find a better solution to our Microservices security problem. Before we dive deep into that, let me just summarize the purpose of each standard and how we can use them.

* OAuth2 - Is an access delegation protocol. The client authenticates with the authorization server and gets an opaque token which is known as 'Access token'. An Access token has zero information about the user/client. It only has a reference to the user information that can only be retrieved by the authorization server. Hence, this is known as a 'by-reference token' and it is safe to use this token even in the public network/internet.
* OpenID Connect behaves similarly to OAuth, but, in addition to the Access token, the authorization server issues an ID token which contains information about the user. This is often implemented by a JWT (JSON Web Token) and that is signed by an authorization server. So, this ensures the trust between the authorization server and the client. JWT token is therefore known as a 'By-value token' as it contains the information of the user and obviously is not safe to use it outside the internal network.

Now, let's see how we can use these standards to secure microservices in our retail example.

Microservice security with OAuth2 and OpenID Connect

As shown in the figure above, these are the key steps involved in implementing microservices security:

* Leave authentication to OAuth and the OpenID Connect server(Authorization Server), so that microservices successfully provide access given someone has the right to use the data.
* Use the API-GW style, in which there is a single entry point for all the client requests.
* The client connects to the authorization server and obtains the Access Token (by-reference token). Then send the access token to the API-GW along with the request.
* Token Translation at the Gateway - API-GW extracts the access token and sends it to the authorization server to retrieve the JWT (by value-token).
* Then GW passes this JWT along with the request to the microservices layer.
* JWTs contains the necessary information to help in storing user sessions, etc. If each service can understand a JSON web token, then you have distributed your identity mechanism which is allowing you to transport identity throughout your system.
* At each microservice layer, we can have a component that processes the JWT, which is a quite trivial implementation.

### Transactions

How about the transactions support in microservices? In fact, supporting distributed transactions across multiple microservices is an exceptionally complex task. The microservice architecture itself encourages the transaction-less coordination between services.

The idea is that a given service is fully self-contained and based on the single responsibility principle. The need to have distributed transactions across multiple microservices is often a symptom of a design flaw in microservice architecture and usually can be sorted out by refactoring the scopes of microservices. However, if there is a mandatory requirement to have distributed transactions across multiple services, then such scenarios can be realized with the introduction of 'compensating operations' at each microservice level. The key idea is, a given microservice is based on the single responsibility principle and if a given microservice failed to execute a given operation, we can consider that as a failure of that entire microservice. Then all the other (upstream) operations have to be undone by invoking the respective compensating operation of those microservices.

### Design for Failures

Microservice architecture introduces a dispersed set of services and, compared to a monolithic design, that increases the possibility of having failures at each service level. A given microservice can fail due to network issues, unavailability of the underlying resources, etc. An unavailable or unresponsive microservice should not bring the whole microservices-based application down. Thus, microservices should be fault-tolerant, be able to recover when that is possible, and the client has to handle it gracefully.

Also, since services can fail at any time, it's important to be able to detect (real-time monitoring) the failures quickly and, if possible, automatically restore the services.

There are several commonly used patterns in handling errors in a Microservices context.

#### Circuit Breaker

When you are doing an external call to a microservice, you configure a fault monitor component with each invocation and when the failures reach a certain threshold then that component stops any further invocations of the service (trips the circuit). After a certain number of requests in open state (which you can configure), change the circuit back to close state.

This pattern is quite useful to avoid unnecessary resource consumption, request delay due to timeouts, and also gives us to chance to monitor the system (based on the active open circuits states).

#### Bulkhead

As microservice application comprises of the number of microservices, the failures of one part of the microservices-based application should not affect the rest of the application. Bulkhead pattern is about isolating different parts of your application, so that a failure of a service in the application does not affect any of the other services.

#### Timeout

The timeout pattern is a mechanism which is allowing you to stop waiting for a response from the microservice when you think that it won't come. Here you can configure the time interval that you wish to wait.

So, where and how do we use these patterns with microservices? In most cases, most of these patterns are applicable at the Gateway level. Which means when the microservices are not available or not responding, at the Gateway level we can decide whether to send the request to the microservice using circuit breakers or timeout pattern. Also, it's quite important to have patterns such as bulkhead implemented at the Gateway level, as it's the single entry point for all the client requests, so a failure in a give service should not affect the invocation of the other microservices.

In addition, Gateway can be used as the central point that we can obtain the status and monitor of each microservice as each microservices is invoked through the Gateway.

### Microservices, Enterprise Integration, API Management, and Beyond.

We have discusse various characteristics of Microservices architecture and how you could implement them in the modern enterprise IT landscape. However, we should keep in mind that Microservices is not a panacea. The blind adaptation of buzzword concepts is not going to solve your 'real' Enterprise IT problems. As you have seen throughout this blog post, there are quite a lot of advantages of microservices and we should leverage. But, we also have to keep in mind that it is not realistic to solve all the enterprise IT problems with microservics. For instance, Microservices architecture promotes to eliminate ESB as the central bus, but when it comes to real world IT, there are quite a lot of existing applications/services which are not based on Microservices. So, to integrate with them, we need some sort of integration bus. So, ideally, a hybrid approach of Microservices and other enterprise architectural concepts such as Integration would be more realistic. I will discuss them further in a separate blog post.

Hope this gives you a much clearer idea of how you can use Microservices in your enterprises.

**Design Patterns for Microservices**

microservice architecture has been built:

1. Scalability
2. Availability
3. Resiliency
4. Independent, autonomous
5. Decentralized governance
6. Failure isolation
7. Auto-Provisioning
8. Continuous delivery through DevOps

Applying all these principles brings several challenges and issues. Let's discuss those problems and their solutions.

## 1. Decomposition Patterns

### a. Decompose by Business Capability

#### **Problem**

Microservices is all about making services loosely coupled, applying the single responsibility principle. However, breaking an application into smaller pieces has to be done logically. How do we decompose an application into small services?

#### **Solution**

One strategy is to decompose by business capability. A business capability is something that a business does in order to generate value. The set of capabilities for a given business depend on the type of business. For example, the capabilities of an insurance company typically include sales, marketing, underwriting, claims processing, billing, compliance, etc. Each business capability can be thought of as a service, except it’s business-oriented rather than technical.

### b. Decompose by Subdomain

#### **Problem**

Decomposing an application using business capabilities might be a good start, but you will come across so-called "God Classes" which will not be easy to decompose. These classes will be common among multiple services. For example, the Order class will be used in Order Management, Order Taking, Order Delivery, etc. How do we decompose them?

#### **Solution**

For the "God Classes" issue, DDD (Domain-Driven Design) comes to the rescue. It uses subdomains and bounded context concepts to solve this problem. DDD breaks the whole domain model created for the enterprise into subdomains. Each subdomain will have a model, and the scope of that model will be called the bounded context. Each microservice will be developed around the bounded context.

**Note**: Identifying subdomains is not an easy task. It requires an understanding of the business. Like business capabilities, subdomains are identified by analyzing the business and its organizational structure and identifying the different areas of expertise.

### c. Strangler Pattern

#### **Problem**

So far, the design patterns we talked about were decomposing applications for greenfield, but 80% of the work we do is with brownfield applications, which are big, monolithic applications. Applying all the above design patterns to them will be difficult because breaking them into smaller pieces at the same time it's being used live is a big task.

#### **Solution**

The Strangler pattern comes to the rescue. The Strangler pattern is based on an analogy to a vine that strangles a tree that it’s wrapped around. This solution works well with web applications, where a call goes back and forth, and for each URI call, a service can be broken into different domains and hosted as separate services. The idea is to do it one domain at a time. This creates two separate applications that live side by side in the same URI space. Eventually, the newly refactored application “strangles” or replaces the original application until finally you can shut off the monolithic application.

## 2. Integration Patterns

### A. API Gateway Pattern

#### **Problem**

When an application is broken down to smaller microservices, there are a few concerns that need to be addressed:

1. How to call multiple microservices abstracting producer information.
2. On different channels (like desktop, mobile, and tablets), apps need different data to respond for the same backend service, as the UI might be different.
3. Different consumers might need a different format of the responses from reusable microservices. Who will do the data transformation or field manipulation?
4. How to handle different type of Protocols some of which might not be supported by producer microservice.

#### **Solution**

An API Gateway helps to address many concerns raised by microservice implementation, not limited to the ones above.

1. An API Gateway is the single point of entry for any microservice call.
2. It can work as a proxy service to route a request to the concerned microservice, abstracting the producer details.
3. It can fan out a request to multiple services and aggregate the results to send back to the consumer.
4. One-size-fits-all APIs cannot solve all the consumer's requirements; this solution can create a fine-grained API for each specific type of client.
5. It can also convert the protocol request (e.g. AMQP) to another protocol (e.g. HTTP) and vice versa so that the producer and consumer can handle it.
6. It can also offload the authentication/authorization responsibility of the microservice.

### b. Aggregator Pattern

#### **Problem**

We have talked about resolving the aggregating data problem in the API Gateway Pattern. However, we will talk about it here holistically. When breaking the business functionality into several smaller logical pieces of code, it becomes necessary to think about how to collaborate the data returned by each service. This responsibility cannot be left with the consumer, as then it might need to understand the internal implementation of the producer application.

#### **Solution**

The Aggregator pattern helps to address this. It talks about how we can aggregate the data from different services and then send the final response to the consumer. This can be done in two ways:

1. A **composite microservice** will make calls to all the required microservices, consolidate the data, and transform the data before sending back.

2. An **API Gateway** can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.

It is recommended if any business logic is to be applied, then choose a composite microservice. Otherwise, the API Gateway is the established solution.

### c. Client-Side UI Composition Pattern

#### **Problem**

When services are developed by decomposing business capabilities/subdomains, the services responsible for user experience have to pull data from several microservices. In the monolithic world, there used to be only one call from the UI to a backend service to retrieve all data and refresh/submit the UI page. However, now it won't be the same. We need to understand how to do it.

#### **Solution**

With microservices, the UI has to be designed as a skeleton with multiple sections/regions of the screen/page. Each section will make a call to an individual backend microservice to pull the data. That is called composing UI components specific to service. Frameworks like AngularJS and ReactJS help to do that easily. These screens are known as Single Page Applications (SPA). This enables the app to refresh a particular region of the screen instead of the whole page.

## 3. Database Patterns

### a. Database per Service

#### **Problem**

There is a problem of how to define database architecture for microservices. Following are the concerns to be addressed:

1. Services must be loosely coupled. They can be developed, deployed, and scaled independently.

2. Business transactions may enforce invariants that span multiple services.

3. Some business transactions need to query data that is owned by multiple services.

4. Databases must sometimes be replicated and sharded in order to scale.

5. Different services have different data storage requirements.

#### **Solution**

To solve the above concerns, one database per microservice must be designed; it must be private to that service only. It should be accessed by the microservice API only. It cannot be accessed by other services directly. For example, for relational databases, we can use private-tables-per-service, schema-per-service, or database-server-per-service. Each microservice should have a separate database id so that separate access can be given to put up a barrier and prevent it from using other service tables.

### b. Shared Database per Service

#### **Problem**

We have talked about one database per service being ideal for microservices, but that is possible when the application is greenfield and to be developed with DDD. But if the application is a monolith and trying to break into microservices, denormalization is not that easy. What is the suitable architecture in that case?

#### **Solution**

A shared database per service is not ideal, but that is the working solution for the above scenario. Most people consider this an anti-pattern for microservices, but for brownfield applications, this is a good start to break the application into smaller logical pieces. This should not be applied for greenfield applications. In this pattern, one database can be aligned with more than one microservice, but it has to be restricted to 2-3 maximum, otherwise scaling, autonomy, and independence will be challenging to execute.

### c. Command Query Responsibility Segregation (CQRS)

#### **Problem**

Once we implement database-per-service, there is a requirement to query, which requires joint data from multiple services — it's not possible. Then, how do we implement queries in microservice architecture?

#### **Solution**

CQRS suggests splitting the application into two parts — the command side and the query side. The command side handles the Create, Update, and Delete requests. The query side handles the query part by using the materialized views. The **event sourcing pattern** is generally used along with it to create events for any data change. Materialized views are kept updated by subscribing to the stream of events.

### d. Saga Pattern

#### **Problem**

When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services? For example, for an e-commerce application where customers have a credit limit, the application must ensure that a new order will not exceed the customer’s credit limit. Since Orders and Customers are in different databases, the application cannot simply use a local ACID transaction.

#### **Solution**

A Saga represents a high-level business process that consists of several sub requests, which each update data within a single service. Each request has a compensating request that is executed when the request fails. It can be implemented in two ways:

1. Choreography — When there is no central coordination, each service produces and listens to another service’s events and decides if an action should be taken or not.
2. Orchestration — an orchestrator (object) takes responsibility for a saga’s decision making and sequencing business logic.

## 4. Observability Patterns

### a. Log Aggregation

#### **Problem**

Consider a use case where an application consists of multiple service instances that are running on multiple machines. Requests often span multiple service instances. Each service instance generates a log file in a standardized format. How can we understand the application behavior through logs for a particular request?

#### **Solution**

We need a centralized logging service that aggregates logs from each service instance. Users can search and analyze the logs. They can configure alerts that are triggered when certain messages appear in the logs. For example, PCF does have Loggeregator, which collects logs from each component (router, controller, diego, etc...) of the PCF platform along with applications. AWS Cloud Watch also does the same.

### b. Performance Metrics

#### **Problem**

When the service portfolio increases due to microservice architecture, it becomes critical to keep a watch on the transactions so that patterns can be monitored and alerts sent when an issue happens. How should we collect metrics to monitor application perfomance?

#### **Solution**

A metrics service is required to gather statistics about individual operations. It should aggregate the metrics of an application service, which provides reporting and alerting. There are two models for aggregating metrics:

* Push — the service pushes metrics to the metrics service e.g. NewRelic, AppDynamics
* Pull — the metrics services pulls metrics from the service e.g. Prometheus

### c. Distributed Tracing

#### **Problem**

In microservice architecture, requests often span multiple services. Each service handles a request by performing one or more operations across multiple services. Then, how do we trace a request end-to-end to troubleshoot the problem?

#### **Solution**

We need a service which

* Assigns each external request a unique external request id.
* Passes the external request id to all services.
* Includes the external request id in all log messages.
* Records information (e.g. start time, end time) about the requests and operations performed when handling an external request in a centralized service.

Spring Cloud Slueth, along with Zipkin server, is a common implementation.

### d. Health Check

#### **Problem**

When microservice architecture has been implemented, there is a chance that a service might be up but not able to handle transactions. In that case, how do you ensure a request doesn't go to those failed instances? With a load balancing pattern implementation.

#### **Solution**

Each service needs to have an endpoint which can be used to check the health of the application, such as /health. This API should o check the status of the host, the connection to other services/infrastructure, and any specific logic.

Spring Boot Actuator does implement a /health endpoint and the implementation can be customized, as well.

## 5. Cross-Cutting Concern Patterns

### a. External Configuration

#### **Problem**

A service typically calls other services and databases as well. For each environment like dev, QA, UAT, prod, the endpoint URL or some configuration properties might be different. A change in any of those properties might require a re-build and re-deploy of the service. How do we avoid code modification for configuration changes?

#### **Solution**

Externalize all the configuration, including endpoint URLs and credentials. The application should load them either at startup or on the fly.

Spring Cloud config server provides the option to externalize the properties to GitHub and load them as environment properties. These can be accessed by the application on startup or can be refreshed without a server restart.

### b. Service Discovery Pattern

#### **Problem**

When Micorservices come into the picture, we need to address a few issues in terms of calling services:

1. With container technology, IP addresses are dynamically allocated to the service instances. Every time the address changes, a consumer service can break and need manual changes.
2. Each service URL has to be remembered by the consumer and become tightly coupled.

So how does the consumer or router know all the available service instances and locations?

#### **Solution**

A service registry needs to be created which will keep the metadata of each producer service. A service instance should register to the registry when starting and should de-register when shutting down. The consumer or router should query the registry and find out the location of the service. The registry also needs to do a health check of the producer service to ensure that only working instances of the services are available to be consumed through it. There are two types of service discovery: client-side and server-side. An example of client-side discovery is Netflix Eureka and an example of server-side discovery is AWS ALB.

### c. Circuit Breaker Pattern

#### **Problem**

A service generally calls other services to retrieve data, and there is the chance that the downstream service may be down. There are two problems with this: first, the request will keep going to the down service, exhausting network resources and slowing performance. Second, the user experience will be bad and unpredictable. How do we avoid cascading service failures and handle failures gracefully?

#### **Solution**

The consumer should invoke a remote service via a proxy that behaves in a similar fashion to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period, all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes normal operation. Otherwise, if there is a failure, the timeout period begins again.

Netflix Hystrix is a good implementation of the circuit breaker pattern. It also helps you to define a fallback mechanism which can be used when the circuit breaker trips. That provides a better user experience.

### d. Blue-Green Deployment Pattern

#### **Problem**

With microservice architecture, one application can have many microservices. If we stop all the services then deploy an enhanced version, the downtime will be huge and can impact the business. Also, the rollback will be a nightmare. How do we avoid or reduce downtime of the services during deployment?

#### **Solution**

The blue-green deployment strategy can be implemented to reduce or remove downtime. It achieves this by running two identical production environments, Blue and Green. Let's assume Green is the existing live instance and Blue is the new version of the application. At any time, only one of the environments is live, with the live environment serving all production traffic. All cloud platforms provide options for implementing a blue-green deployment. For more details on this topic, check out [this article](https://dzone.com/articles/blue-green-deployment-for-cloud-native-application).

There are many other patterns used with microservice architecture, like Sidecar, Chained Microservice, Branch Microservice, Event Sourcing Pattern, Continuous Delivery Patterns, and more.

## Need for Event Driven Architecture

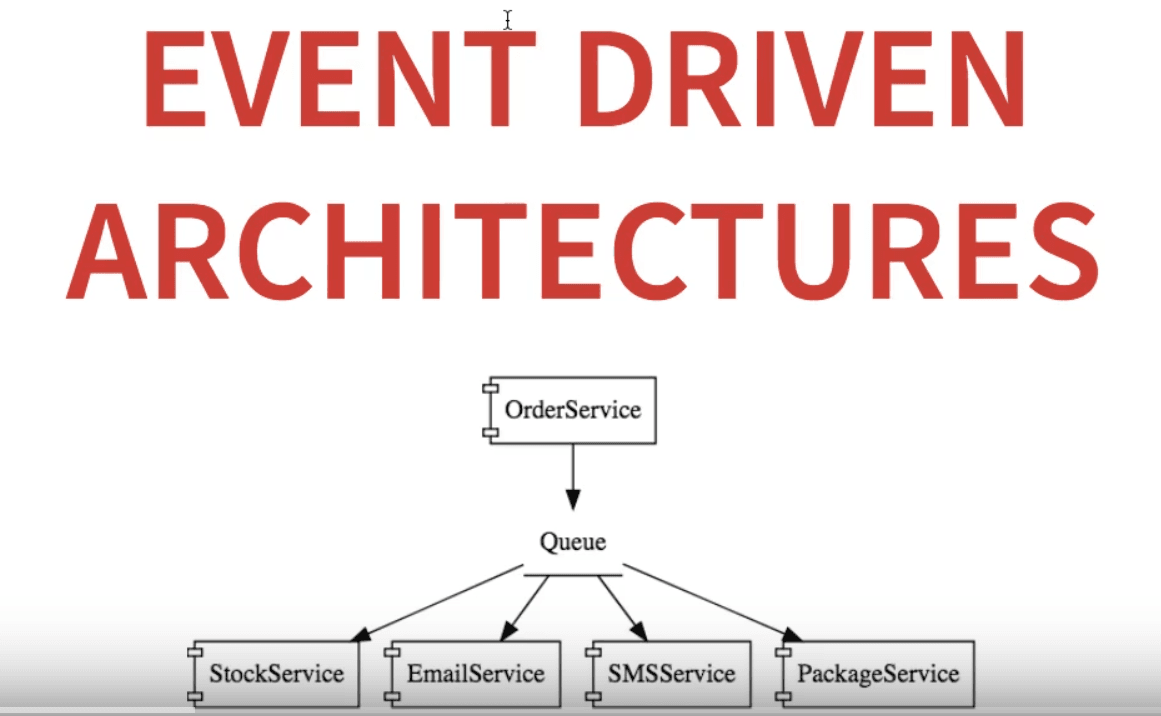
### Use Case

Consider the use case of an online shopping application. Whenever there is an order, it is received by the order service. There are several actions that need to be carried out in order service:

* Update the stock.
* Send out email and SMS.
* Notify the packaging team.

#### Microservices Approach

Alternatively, these are fairly independent activities, and the entire application can be structured to have microservices for them, in a straightforward manner.



In this approach, you create an order event for the request coming in, and place it in the Queue. The rest of the individual services listen in to the queue for order events, and do the processing subsequently.

#### How to Choose?

**How do we make the decision between these two application(monolithic/Micro services architectures?**

The nature of your application decides this.

If your system does not have a very high load at any given time, and also does not have any pressing scalability requirements, then you might want go with a monolith approach.

However, when your system is a large, handling millions of requests every day, and also has very stringent scalability requirements, you might be better off with the microservices approach.

# Advantages of Event Driven Architectures

### Improved Flexibility and Maintainability

One of the most important needs of an application is maintainability. Ease of maintainability comes with the proper **separation of concerns**.

In our example above, the OrderService is only responsible for collecting the order and placing it in the queue. It does not worry about how it is going to be processed, who is going to process it, and so on.Similarly, the StockService is only responsible for updating the stock, and the same with the other microservices.

If there is a need for an additional processing step on an order, you write a new microservice to listen on the queue, and easily integrate it into the system.Such an architecture is clearly extensible, and also easily maintainable, due to separation of concerns.

### **High Scalability**

Let's consider an example.

One fine day, a large volume of emails need to be sent out. In that case, you have the freedom to create a large number of instances of the EmailService to handle additional load. A similar thing can be done to the other microservices in the mix, depending on your needs.

This is a clear advantage of this architecture, over using a single component to handle all the functionality.

### **Improved Availability**

Let's say one of the services listening for order events from the queue, such as the PackageService, goes down.

In a monolithic approach, any one functionality going down would mean the application cannot process orders any more.

In the case of an event driven architecture, the PackageService going down would not prevent the OrderService from putting the order event into the Queue. The OrderService can notify the user that their request was received successfully.

A notification request would then be sent to the troubleshooting team about the PackageServer going down, and while it is being restored, the order event remains in the Queue. It can be processed by all the other services during this time, and when PackageService comes back up, it can process the event as if nothing untoward has happened.

### **Good Reliability**

Event driven architectures ensure a good standard of reliability for the system as a whole. However, the individual microservices can function with different levels of reliability. For example the StockService normally needs to ensure a high level of reliability, the EmailService and SMSService a medium level of reliability, and the PackageService — between low and medium levels of reliability.

### **Good Performance and Responsiveness**

Note that in our example event driven architecture, all that the OrderService does is receive the order and place it on the queue, before acknowledging the user. The user does not need to wait till all the steps are performed on an order. This has high high responsiveness, and is seen by the user as high performance.

# CACHE MANAGEMENT

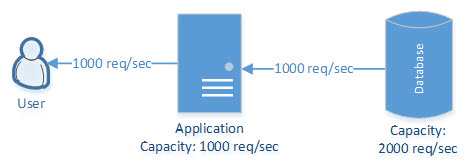
## Concept of the Cache

A cache is a memory buffer used to temporarily store frequently accessed data. It improves performance since data does not have to be retrieved again from the original source. Caching is actually a concept that has been applied in various areas of the computer/networking industry for quite some time, so there are different ways of implementing cache depending upon the use case. In fact, devices such as routers, switches, and PCs use caching to speed up memory access. Another very common cache, used on almost all PCs, is the web browser cache for storing requested objects so that the same data doesn't need to be retrieved multiple times. In a distributed JEE application, the client/server side cache plays a significant role in improving application performance. The client-side cache is used to temporarily store the static data transmitted over the network from the server to avoid unnecessarily calling to the server. On the other hand, the server-side cache is used to store data in memory fetched from other resources.

Cache can be designed on single/multiple JVM or clustered environment. Different scalability scenarios where caching can be used to meet the nonfunctional requirement are as follows.

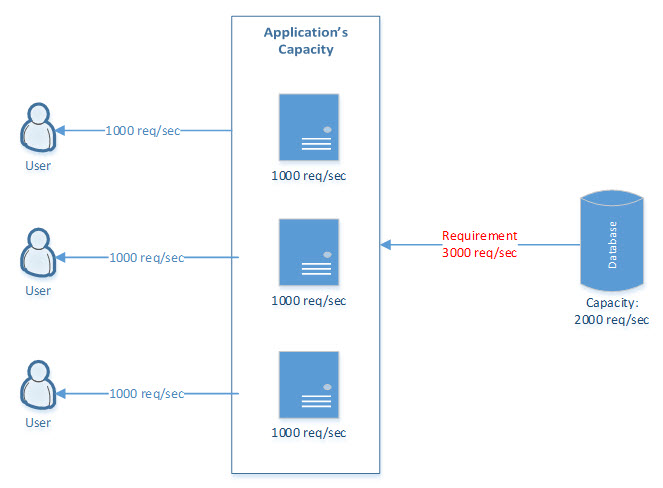
### Vertical Scalability

This can be achieved by upgrading a single machine with more efficient resources (CPU, RAM, HDD, and SSD) and implementing caching. But it has limitations in regards to upgrading a cache up to a certain limit. In the below use case, the application’s performance can be enhanced by adding more memory and implementing caching at the application level.



### Horizontal Scalability

This can be achieved by adding more machines and implementing caching at the application level on each machine. But is still has limitations in regards to communicating with downstream applications where additional servers cannot be added. In the below use case, the overall application’s performance can be enhanced by adding a server/cache per application. The database has some limitations in regards to fulfilling this requirement, but this can be mitigated by storing static/master data in a cache.

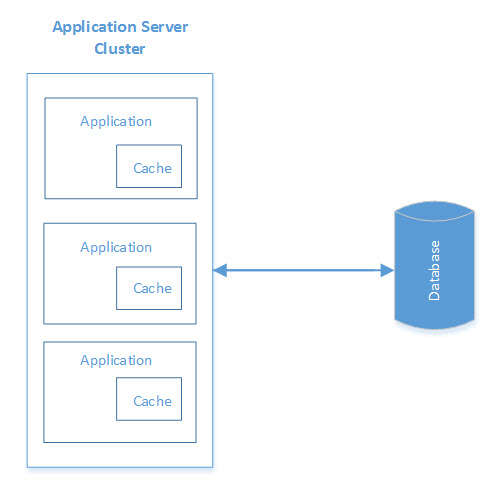


### In-Process Caching

In-process caching enables objects to be stored in the same instance as the application, i.e. the cache is locally available to the application and shares the same memory space.

Here are some important points for considering in-process caching:

* If the application is deployed only in one node, i.e. has a single instance, then in-process caching is the right candidate to store frequently accessed data with fast data access.
* If the in-process cache will be deployed in multiple instances of the application, then keeping data in sync across all instances could be a challenge and cause data inconsistency.
* If server configurations are limited, then this type of cache can degrade the performance of any application since it shares the same memory and CPU. A garbage collector will be invoked often to clean up objects that may lead to performance overhead. If data eviction isn't managed effectively, then out-of-memory errors can occur.

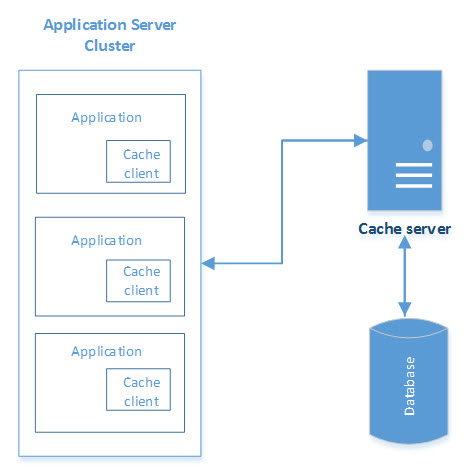


### In-Memory Distributed Caching

Distributed caches (key/value objects) can be built externally to an application that supports read/write to/from data repositories, keeps frequently accessed data in RAM, and avoid continuous fetching data from the data source. Such caches can be deployed on a cluster of multiple nodes, forming a single logical view. Caching clients use hashing algorithms to determine the location of an object in a cluster node.

Here are some important points for considering distributed caching:

* An in-memory distributed cache is the best approach for mid- to large-sized applications with multiple instances on a cluster where performance is key. Data inconsistency and shared memory aren't matters of concern, as a distributed cache is deployed in the cluster as a single logical state.
* As inter-process is required to access caches over a network, latency, failure, and object serialization are some overheads that could degrade performance.
* Its implementation is more difficult than in-process caching.



### In-Memory Database

This type of database is also called a main memory database. It's where data is stored in RAM instead of a hard disk to enable faster responses. Data is stored in compressed format with good SQL support. Relevant database drivers can be used in place of an existing RDBMS.

Replacing the RDBMS with an in-memory database will improve the performance of an application without changing the application layer. Only vertical scaling is possible for scaling up the in-memory database.

### In-Memory Data Grid

This distributed cache solution provides fast access to frequently used data. Data can be cached, replicated, and partitioned across multiple nodes.

Implementing the in-memory data grid will improve the performance of and scale an application without changing the RDBMS.

Key features include:

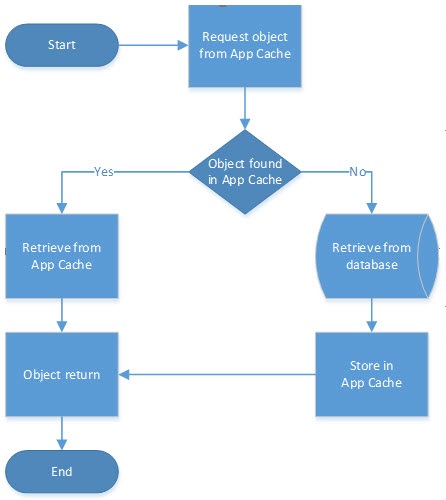
* Parallel computation of the data in memory
* Search, aggregation, and sorting of the data in memory
* Transactions management in memory
* Event-handling

## Cache Use Cases

There are use cases where a cache is used to improve application performance via various commercial/open-source cache frameworks that can be configured in any enterprise application. Here are common cache use cases.

### Application Cache

An application cache is a local cache that an application uses to keep frequently accessed data in memory. An application cache evicts entries automatically to maintain its memory footprint.



### Level 1 (L1) Cache

This is the default transactional cache per session. It can be managed by any Java persistence framework (JPA) or object-relational mapping (ORM) tool.

The L1 cache stores entities object that fall under a specific session and is cleared once a session is closed. If there are multiple transactions inside one session, all entities will be stored from all these transactions.

### Level 2 (L2) Cache

The L2 cache can be configured to provide custom caches that can hold onto the data for all entities to be cached. It can be related to properties, associations, and collections. It's configured at session factory-level and exists as long as the session factory is available.

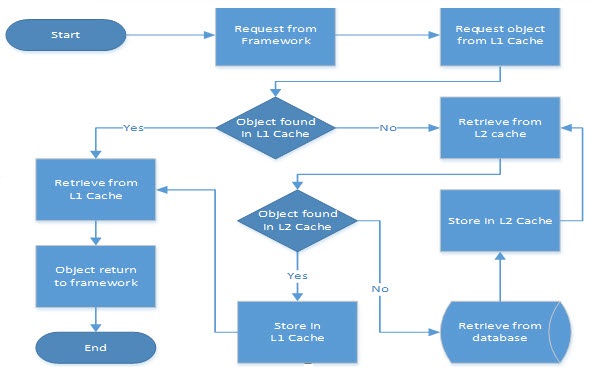
The L2 cache can be configured to be available across:

* Sessions in an application.
* Applications on the same servers with the same database.
* Clusters for different applications on different servers on the same database.

Usages of an L1/L2 cache, if the L2 is configured:

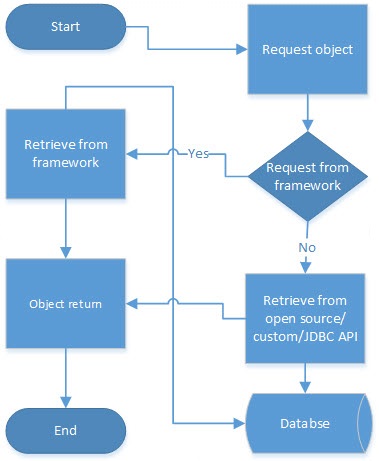
* The standard ORM framework first looks up an entity in the L1 cache, and then the L2 cache. The L1 cache is the initial search space to look up entities. If the cached copy of an entity is found, then it is returned.
* If no cached entity is found in the L1 cache, then it's looked up in the L2 cache.
* If a cached entity is found in the L2 cache, then it's stored in the L1 cache and then returned.
* If an entity is not found in L1 nor L2, then it’s fetched from the database and stored in both caches before returning to the caller.
* L2 cache validates/refreshes itself when any modification happens on entities by any session.
* If the database is modified at all by external processes, i.e. without application session, then the L2 cache cannot be refreshed implicitly until some cache refresh policy is implemented either through the framework API or some custom API.

The following communication diagram illustrates using an L1/L2 cache:



### Hybrid Cache

A hybrid cache is a combination of a cache provided by standard ORM framework and open-source/custom/JDBC API implementations. An application can use a hybrid cache to leverage cache capability that's limited to standard ORM framework. This kind of cache is used in mission-critical applications where response time is significant.



## Caching Design Considerations

Caching design considerations include data loading/updating, performance/memory size, eviction policy, concurrency, and cache statistics.

### Data Loading/Updating

Data loading into a cache is an important design decision to maintain consistency across all cached content. The following approaches can be considered to load data:

* Using default function/configuration provided by standard ORM framework, i.e. Hibernate or OpenJPA.
* Implementing key-value maps using open-source cache APIs, i.e. Google Guava or COTS products like Coherence, Ehcache, or Hazelcast.
* Programmatically loading entities through automatic or explicit insertion.
* External application through synchronous or asynchronous communication.

### Performance/Memory Size 32/64 Bit

Available memory is an important factor to achieve performance SLA and it depends on 32/64 bit JRE, which is further dependent on 32/64-bit CPU architecture machines. In a 32-bit system/JRE, ~1.5 GB of heap is left for application use, while in a 64-bit system/JRE, heap size is dependent on RAM size.

High availability of memory does have cost at runtime and can have negative consequences.

* 30-50% more heap is required on 64-bit compared to 32-bit due to memory layout.
* Maintaining more heap requires more GC work for cleaning unused objects that can degrade performance. Fine-tuning GC can be an option to limit GC pauses.

### Eviction Policy

An eviction policy enables a cache to ensure that the size of the cache doesn't exceed the maximum limit. To achieve this, existing elements are removed from a cache depending on the eviction policy, but it can be customized as per application requirements. There are various popular eviction algorithms used in cache solution:

* Least Recently Used (LRU)
* Least Frequently Used (LFU)
* First In, First Out (FIFO)

### Concurrency

Concurrency is a common issue in enterprise applications. It creates conflict and leaves the system in an inconsistent state. It can happen when multiple clients try to update the same data object at the same time during cache refresh. A common solution is to use a lock, but this may affect performance. Hence, optimization techniques should be considered.

### Cache Statistics

Cache stats help identify the health of the cache and provide information about cache behavior and performance. In general, the following attributes can be used in cache statistics:

* **Hit count**: Number of look-ups encountered when object found
* **Miss count**: Number of look-ups encountered when object not found
* **Load success count**: Number of successfully loaded entries
* **Total load time**: Total time in loading an element
* **Load exception count**: Number of exceptions thrown while loading an entry
* **Eviction count**: Number of entries evicted from the cache

## Summary: Various Cache Solutions

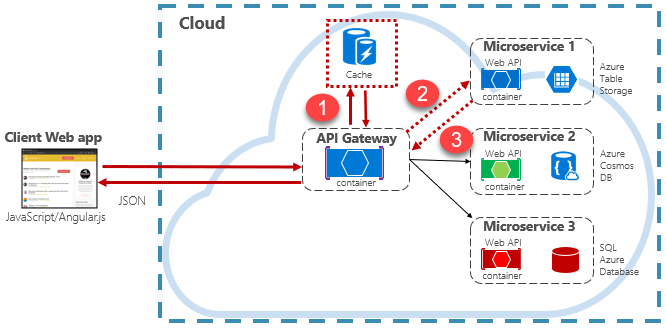
There are various Java caching solutions available — the right choice for you depends on your use case. Here are some questions and comparisons that can assist in identifying the most cost-effective and feasible caching solution for you.

* Do you need a light-weighted or full-fledged cache solution?
* Do you need an open-source, commercial, or framework-provided cache solution?
* Do you need in-process or distributed caching?
* What’s the trade-off between consistency and latency requirements?
* Do you need to maintain the cache for transactional/master data?
* Do you need a replicated cache?
* What about performance, reliability, scalability, and availability?

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Ehcache** | **Hazelcast** | **Redis/Memcached** | **Google Guava** | **Coherence** |
| **Cache type** | Distributed, Level 2 | Distributed data grid, Level 2 | Distributed in-memory store | In-process | Distributed |
| **Open source** | Yes | Yes | Yes | Yes | No |
| **Replicated** | Yes | Yes | Yes | No | Yes |
| **(JSR-107)-compliant** | Yes | Yes | Yes | - | Yes |
| **Configuration complexity** | Easy | Easy | Medium | Easy | Complex |
| **Eviction algorithms** | LRU, FLU | LRU, FLU | LRU | LRU | LRU, LFU, hybrid, and custom |
| **Clustering support** | Yes | Yes | Yes | No | Yes |

## Caching architecture

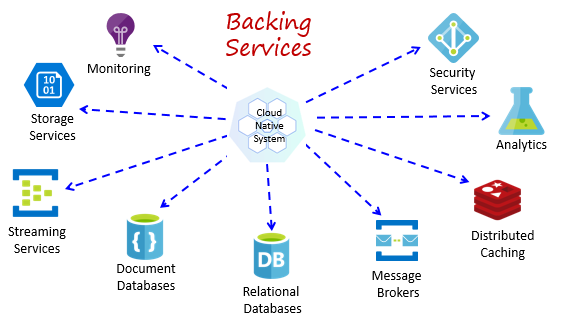
Cloud native applications typically implement a **distributed caching architecture**. The cache is hosted as a cloud-based [backing service](https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition#backing-services), separate from the microservices.



**Figure 5-15**: Caching in a cloud native app

### Backing services

Cloud-native systems depend upon many different ancillary resources, such as data stores, message brokers, monitoring, and identity services. These services are known as [backing services](https://12factor.net/backing-services).



In the previous figure, note how the cache is independent of and shared by the microservices. In this scenario, the cache is invoked by the [API Gateway](https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/front-end-communication). As discussed in chapter 4, the gateway serves as a front end for all incoming requests. The distributed cache increases system responsiveness by returning cached data whenever possible. Additionally, separating the cache from the services allows the cache to scale up or out independently to meet increased traffic demands.

The previous figure presents a common caching pattern known as the [cache-aside pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/cache-aside). For an incoming request, you first query the cache (step #1) for a response. If found, the data is returned immediately. If the data doesn't exist in the cache (known as a [cache miss](https://www.techopedia.com/definition/6308/cache-miss)), it's retrieved from a local database in a downstream service (step #2). It's then written to the cache for future requests (step #3), and returned to the caller. Care must be taken to periodically evict cached data so that the system remains timely and consistent.

As a shared cache grows, it might prove beneficial to partition its data across multiple nodes. Doing so can help minimize contention and improve scalability. Many Caching services support the ability to dynamically add and remove nodes and rebalance data across partitions. This approach typically involves clustering. Clustering exposes a collection of federated nodes as a seamless, single cache. Internally, however, the data is dispersed across the nodes following a predefined distribution strategy that balances the load evenly.

## Azure Cache for Redis

[Azure Cache for Redis](https://azure.microsoft.com/services/cache/) is a secure data caching and messaging broker service, fully managed by Microsoft. Consumed as a Platform as a Service (PaaS) offering, it provides high throughput and low-latency access to data. The service is accessible to any application within or outside of Azure.

The Azure Cache for Redis service manages access to open-source Redis servers hosted across Azure data centers. The service acts as a facade providing management, access control, and security. The service natively supports a rich set of data structures, including strings, hashes, lists, and sets. If your application already uses Redis, it will work as-is with Azure Cache for Redis.

Azure Cache for Redis is more than a simple cache server. It can support a number of scenarios to enhance a microservices architecture:

* An in-memory data store
* A distributed non-relational database
* A message broker
* A configuration or discovery server

For advanced scenarios, a copy of the cached data can be [persisted to disk](https://docs.microsoft.com/en-us/azure/azure-cache-for-redis/cache-how-to-premium-persistence). If a catastrophic event disables both the primary and replica caches, the cache is reconstructed from the most recent snapshot.

# Cache-Aside pattern

Load data on demand into a cache from a data store. This can improve performance and also helps to maintain consistency between data held in the cache and data in the underlying data store.

## Context and problem

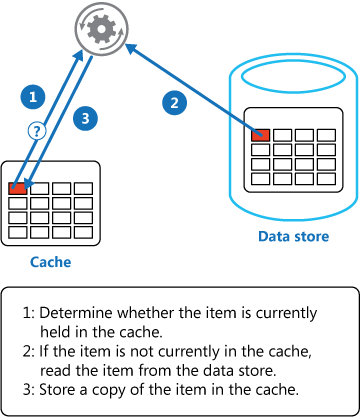
Applications use a cache to improve repeated access to information held in a data store. However, it's impractical to expect that cached data will always be completely consistent with the data in the data store. Applications should implement a strategy that helps to ensure that the data in the cache is as up-to-date as possible, but can also detect and handle situations that arise when the data in the cache has become stale.

## Solution

Many commercial caching systems provide read-through and write-through/write-behind operations. In these systems, an application retrieves data by referencing the cache. If the data isn't in the cache, it's retrieved from the data store and added to the cache. Any modifications to data held in the cache are automatically written back to the data store as well.

For caches that don't provide this functionality, it's the responsibility of the applications that use the cache to maintain the data.

An application can emulate the functionality of read-through caching by implementing the cache-aside strategy. This strategy loads data into the cache on demand. The figure illustrates using the Cache-Aside pattern to store data in the cache.



If an application updates information, it can follow the write-through strategy by making the modification to the data store, and by invalidating the corresponding item in the cache.

When the item is next required, using the cache-aside strategy will cause the updated data to be retrieved from the data store and added back into the cache.

## Issues and considerations

Consider the following points when deciding how to implement this pattern:

**Lifetime of cached data**. Many caches implement an expiration policy that invalidates data and removes it from the cache if it's not accessed for a specified period. For cache-aside to be effective, ensure that the expiration policy matches the pattern of access for applications that use the data. Don't make the expiration period too short because this can cause applications to continually retrieve data from the data store and add it to the cache. Similarly, don't make the expiration period so long that the cached data is likely to become stale. Remember that caching is most effective for relatively static data, or data that is read frequently.

**Evicting data**. Most caches have a limited size compared to the data store where the data originates, and they'll evict data if necessary. Most caches adopt a least-recently-used policy for selecting items to evict, but this might be customizable. Configure the global expiration property and other properties of the cache, and the expiration property of each cached item, to ensure that the cache is cost effective. It isn't always appropriate to apply a global eviction policy to every item in the cache. For example, if a cached item is very expensive to retrieve from the data store, it can be beneficial to keep this item in the cache at the expense of more frequently accessed but less costly items.

**Priming the cache**. Many solutions prepopulate the cache with the data that an application is likely to need as part of the startup processing. The Cache-Aside pattern can still be useful if some of this data expires or is evicted.

**Consistency**. Implementing the Cache-Aside pattern doesn't guarantee consistency between the data store and the cache. An item in the data store can be changed at any time by an external process, and this change might not be reflected in the cache until the next time the item is loaded. In a system that replicates data across data stores, this problem can become serious if synchronization occurs frequently.

**Local (in-memory) caching**. A cache could be local to an application instance and stored in-memory. Cache-aside can be useful in this environment if an application repeatedly accesses the same data. However, a local cache is private and so different application instances could each have a copy of the same cached data. This data could quickly become inconsistent between caches, so it might be necessary to expire data held in a private cache and refresh it more frequently. In these scenarios, consider investigating the use of a shared or a distributed caching mechanism.

The following table highlights the Twelve-Factor methodology:

| **THE TWELVE-FACTOR APPLICATION** | | |
| --- | --- | --- |
|  | **Factor** | **Explanation** |
| 1 | Code Base | A single code base for each microservice, stored in its own repository.  Tracked with version control, it can deploy to multiple environments (QA, Staging, Production). |
| 2 | Dependencies | Each microservice isolates and packages its own dependencies,  embracing changes without impacting the entire system. |
| 3 | Configurations | Configuration information is moved out of the microservice and externalized through a configuration management tool outside of the code. The same deployment can propagate across environments with the correct configuration applied. |
| 4 | Backing Services | Ancillary resources (data stores, caches, message brokers) should be exposed via an addressable URL. Doing so decouples the resource from the application, enabling it to be interchangeable. |
| 5 | Build, Release, Run | Each release must enforce a strict separation across the build, release, and run stages. Each should be tagged with a unique ID and support the ability to roll back. Modern CI/CD systems help fulfill this principle. |
| 6 | Processes | Each microservice should execute in its own process, isolated from other running services. Externalize required state to a backing service such as a distributed cache or data store. |
| 7 | Port Binding | Each microservice should be self-contained with its interfaces and functionality exposed on its own port. Doing so provides isolation from other microservices. |
| 8 | Concurrency | Services scale out across a large number of small identical processes (copies) as opposed to scaling-up a single large instance on the most powerful machine available. |
| 9 | Disposability | Service instances should be disposable, favoring fast startups to increase scalability opportunities and  graceful shutdowns to leave the system in a correct state. Docker containers along with an orchestrator inherently satisfy this requirement. |
| 10 | Dev/Prod Parity | Keep environments across the application lifecycle as similar as possible, avoiding costly shortcuts.  Here, the adoption of containers can greatly contribute by promoting the same execution environment. |
| 11 | Logging | Treat logs generated by microservices as event streams. Process them with an event aggregator  and propagate the data to data-mining/log management tools like Azure Monitor or Splunk and  eventually long-term archival. |
| 12 | Admin Processes | Run administrative/management tasks as one-off processes. Tasks can include data cleanup and pulling analytics for a report. Tools executing these tasks should be invoked from the production environment, but separately from the application. |
|  |  |  |

In the book, [Beyond the Twelve-Factor App](https://content.pivotal.io/blog/beyond-the-twelve-factor-app), author Kevin Hoffman details each of the original 12 factors (written in 2011). Additionally, he discusses three additional factors that reflect today's modern cloud application design.

| **TABLE 3** | | |
| --- | --- | --- |
|  | **New Factor** | **Explanation** |
| 13 | API First | Make everything a service. Assume your code will be consumed by a front-end client, gateway, or another service. |
| 14 | Telemetry | On a workstation, you have deep visibility into your application and its behavior. In the cloud, you don't. Make sure your design includes the collection of monitoring, domain-specific, and health/system data. |
| 15 | Authentication/ Authorization | Implement identity from the start. Consider [RBAC (role-based access control)](https://docs.microsoft.com/en-us/azure/role-based-access-control/overview) features available in public clouds. |