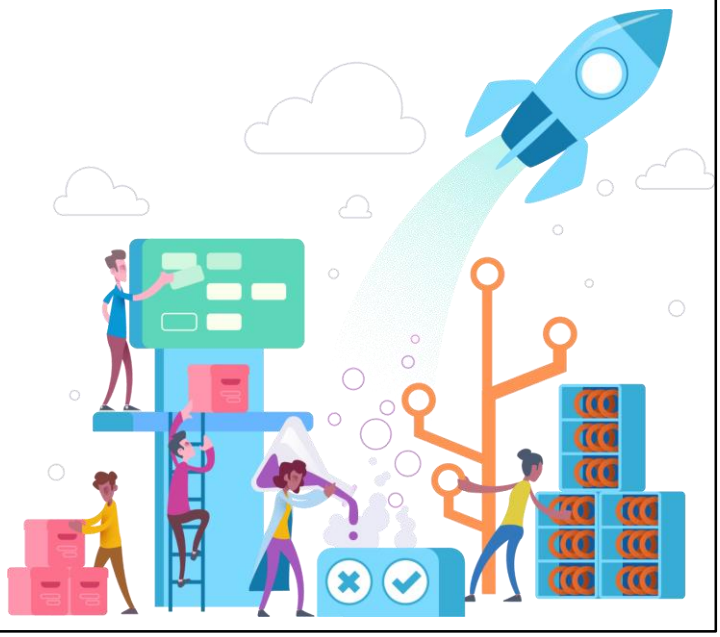
Microsoft

Microservices

Alex Mang
05/27/2019



SPEAKER NOTES:
This slide deck may support educating the customers about Microservices. The target audience are customer architects and their senior engineers (Level 300/400). It is mostly cloud agnostic, but Microsoft Azure can help a lot managing all this moving pieces. Microservices itself are quiet useful during app modernization and cloud migration. Mastering microservices requires a high degree of automation, so additional education in Azure DevOps might be a possible follow-up to this presentation. Finally looking into container orchestrators like Azure Kubernetes Service or Service Fabric will also be required for adopting microservices in an efficient manner.

Microsoft

Microservices

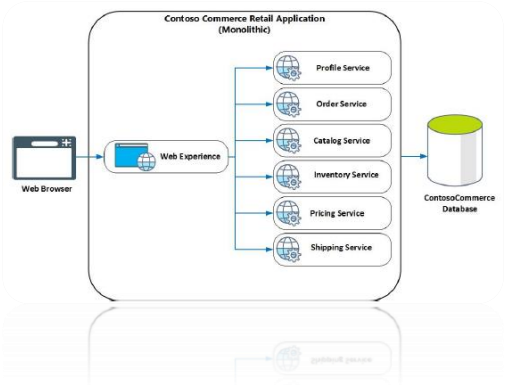
From Wikipedia, the free encyclopedia

Some of this article's listed sources **may not be reliable**. Please help this article by looking for better, more reliable sources. Unreliable citations may be challenged or deleted. *(October 2018) (Learn how and when to remove this template message)*

Microservices are a software development technique—a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. In a microservices architecture, services are fine-grained and the protocols are lightweight. The benefit of decomposing an application into different smaller services is that it improves modularity. This makes the application easier to understand, develop, test, and become more resilient to architecture erosion.[1] It parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently.[2] It also allows the architecture of an individual service to emerge through continuous refactoring.[3] Microservices-based architectures enable continuous delivery and deployment.[4]

Recap: Monoliths

- Services implement multiple business concerns
- Services communicate with a single database
- Application affected by single database schema change
- Services are packaged and deployed as a single unit
- Service change requires application redeployment
- Application can only be scaled as a whole
- Unstable service can bring whole application down



SPEAKER NOTES:
In order to better understand the microservices, someone also has to understand the characteristics of monoliths and see how they differ. Monoliths basically are packaged and deployed as a single unit, implementing multiple business concerns. In contrast, microservices split this business concerns on multiple packaged and deployable units. As we will see, microservices are no silver bullet. There are use cases for microservices and for monoliths as well.

Microservices

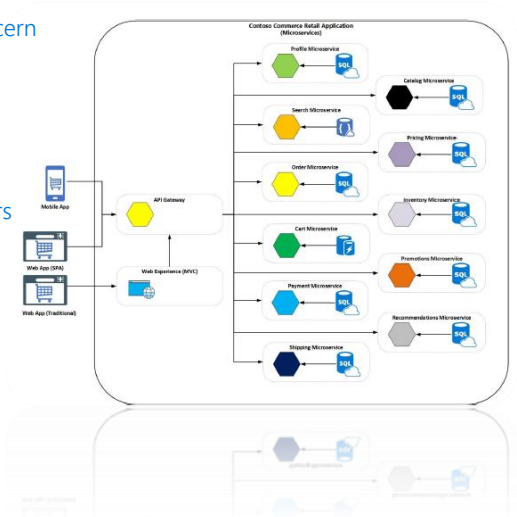
Definition

> The microservice architectural style is an approach to developing a single application as a **suite of small services**, each **running in its own process** and communicating with lightweight mechanisms, often an HTTP resource API. These services are **built around business capabilities** and **independently deployable** by fully automated deployment machinery. There is a **bare minimum of centralized management** of these services, which may be written in different programming languages and use different data storage.
>
> - James Lewis & Martin Fowler

SPEAKER NOTES:
There are many definitions for microservices, so we just picked the most prominent one by James Lewis and Martin Fowler. Generally speaking, microservices follow a rather holistic approach so they can evolve individually, basically by aligning business, services and team boundaries. Since microservices are independent to each other with no need for central management, they can also be managed and scaled individually. This especially helps you when scaling towards multiple distributed teams, which is one of the main use cases for microservices.

Characteristics

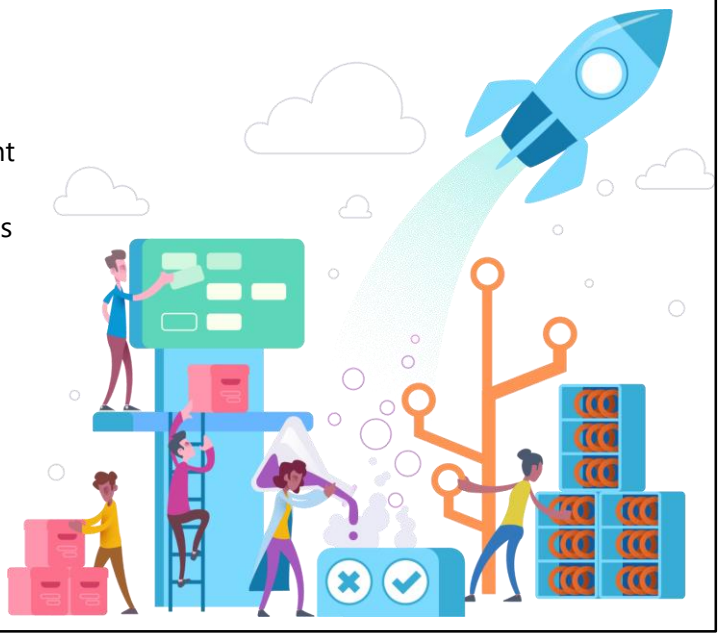
- Services implement single business or technical concern (single responsibility principle)
- Services are developed, deployed and scaled individually
- Services run in their own processes and/or containers
- Each service has its own data repository
- Clients access the services through well-defined APIs managed by API Gateway



SPEAKER NOTES:
This details the previous slides and highlights the microservices characteristics. This slides can be seen in contrast to the slide about monoliths. While monoliths implement multiple business concerns, each microservices implement a single business or technical concern. In order to develop, deploy and scale each microservice individually, microservices run in their own process/container and have their own data repository. Finally highlight that you also require some infrastructure pieces like API Gateways and container orchestrators, which Azure can provide you out of the box.

Benefits

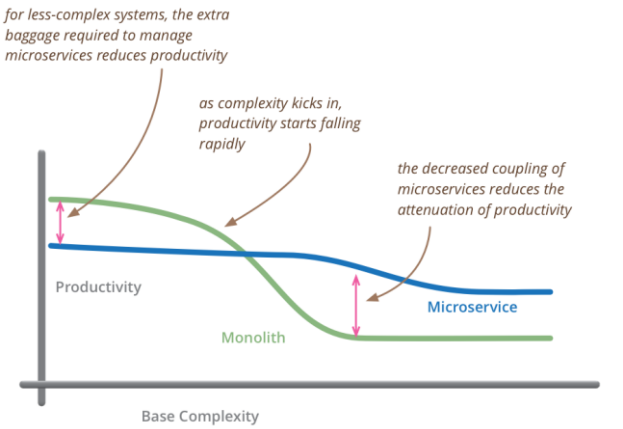
- ❑ Independent development
- ❑ Independent deployments
- ❑ Small, focused teams
- ❑ Mixed technology stacks
- ❑ Fault isolation
- ❑ Granular scaling



SPEAKER NOTES:
Microservices were vastly adopted, since they come also with a lot of benefits. They might enable other best practices i.e. doing DevOps with a large monolith might be quiet challenging. Microservices are much smaller and independently deployable, so DevOps adoption is much easier. Better have many speed boats compared to a large tanker. Also mixing technology stacks is quiet common these days, i.e. some AI/ML experts can write some custom machine learning and expose it as a microservice. The consuming engineering teams do not have to know machine learning at all, they just consume the microservice via a convenient API. That's what Microsoft internally did for instance with the Cognitive Services, but also Azure DevOps is built on top of microservices.

Challenges

- Operational Complexity
- Service boundaries
- Data consistency & integrity
- Service discovery
- Network congestion & latency
- DevOps

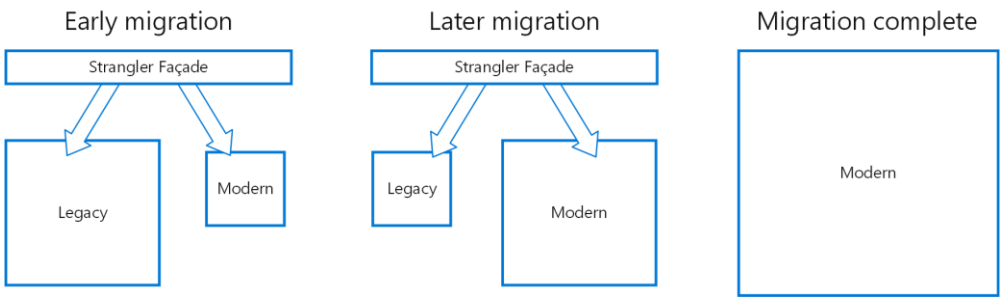


- Martin Fowler

but remember the skill of the team will outweigh any monolith/microservice choice

SPEAKER NOTES:
As we will see, microservices are no silver bullet. They come with a lot of operational complexity to master. Basically there is a break even, when it is worth to use microservices over monoliths – see graph on the right. When adopting microservices, the challenges listed must be individual addressed and we will talk about them in a minute. That's why Azure is a great choice for hosting microservices, considering most of the required capabilities are provided out of the box. Take for instance Azure DevOps, which greatly helps you automating your deployment. Container orchestrators like Azure Kubernetes Service also help to shift the complexity barrier to the left. Both also work quiet well with Azure Monitoring, so you always know what is going on within your microservices. In contrast imagine doing microservices on premise using virtual machines., which requires a lot of additional engineering on top of them.

Breaking up the monolith

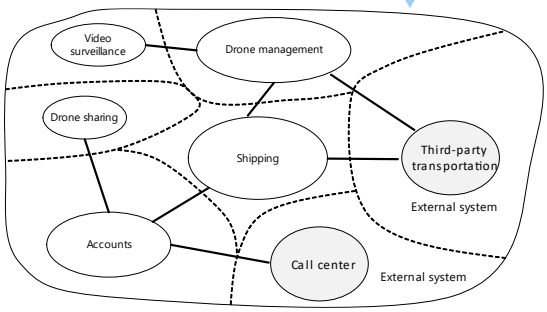


Strangler pattern

SPEAKER NOTES:
Basically there are two approaches to adopt microservices – starting green-field or brown-field. Green-field refers to adopting microservices when developing a new application from scratch. Very common is also adopting microservices during app modernization of an existing application – the so called brown-field approach. You cannot move a legacy monolithic app to a microservices approach at once - you have to do it step by step. Use a Strangler Façade to provide a stable interface to the client, while you do the migration. Basically the client should not know if it talks to the legacy app or to the modern microservices. Slowly extract and move bits of your legacy app into your modern microservices with the Strangler Façade redirecting the client calls. When everything is shifted over to the modern microservices architecture, just remove the Strangler Façade and you are done.

Service boundaries

Analyze domain → Define bounded contexts → Define entities, aggregates & services → Identity microservices



> A **bounded context** is simply the boundary within a domain where a particular domain model applies.
>
> - Domain-driven Design by Eric Evans

SPEAKER NOTES:
As stated previously, microservices follow a rather holistic approach so they can evolve individually, basically by aligning business, services and team boundaries. But how to identify the business boundaries? How can you derive the service boundaries from them? This is where the well-established Domain-driven Design or short DDD by Eric Evans comes into play. With DDD you first identify sub domain within your business domains, the so-called bounded context. Take an e-commerce application for instance: Ordering, shipping, catalogue, etc. Usually you talk to different domain experts, each one using their own language. This deferent domain experts and their languages or so-called ubiquitous language are good indicators for bounded contexts. A microservice itself should not span more than one bounded context, since bounded contexts should evolve independently. If the bounded context is still too big, take a look into other DDD concepts like aggregates and domain services as well. You can find more detailed information about them in the links provided at the end. We will not go into detail here since you can fill easily a whole presentation about DDD. There are also alternatives to using DDD. Event-storming invented by Alberto Brandolini is also a quiet common technique to better understand your business boundaries. Finally encapsulating non-functional requirements like

authentication in a separate microservice makes a lot of sense as well.

Team boundaries

> Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.
>
> - Melvin E. Conway

Fine-grained communication required for developing a single microservice, following choose small co-located team who owns a microservice („*2 pizza team*“)

Coarse-grained communication desired between loosely coupled teams in order to scale, following no requirement for co-location but for stable and versioned APIs

Align domain, service and team boundaries

SPEAKER NOTES:
So we aligned the service boundaries to the business boundaries, but what about the team structure. Melvin E. Conway formulated a law named after him to outline the implications of organizations and their communication structure on the product designs they produce. Co-located teams tend to couple their product more, since they can communicate frequently and often. Split teams try to agree on stable APIs, since they cannot communicate in a so fine-grained fashion. This principle can be leveraged when developing microservices. While you want coarse-grained communication via stable APIs between microservices, you want fine-grained communication within your microservices since you interact on code-level. Following let a co-located team own a microservice and only them, with other teams located anywhere only use stable and versioned APIs to use this microservice. As a result, we aligned domain, service and team boundaries, which is quiet powerful to evolve these microservices and their bounded context individually.

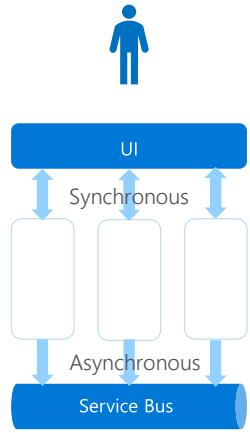
Interservice communication

Synchronous Communication

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. Typically RESTful APIs are external-facing or exposed towards the UI. Callers must implement retry and timeouts for resiliency. APIs must be semantically versioned.

Asynchronous Communication

Asynchronous message passing. In this pattern, a service sends messages without waiting for a response, and one or more services process the message asynchronously. Typically inter-service communication is implemented via asynchronous messaging using AMQP or similar protocols. Resilience is built into the underlying service bus, which follows the paradigm "smart endpoints and dump pipes". Messages must be semantically versioned.

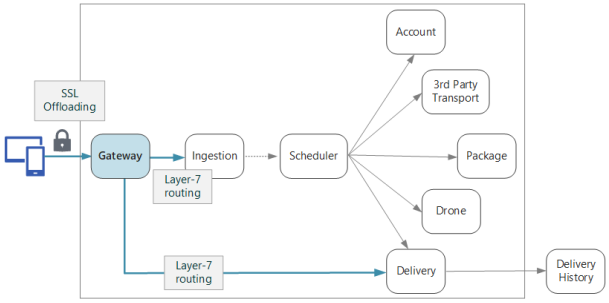


SPEAKER NOTES:
Let's make an analogy when comparing synchronous to asynchronous communication. Synchronous communication is like paying with your credit card. You insert your card, input your PIN and wait for the transaction to be approved. In contrast, Asynchronous communication is non-blocking. It is more like ordering a pizza. You order it online and, at some point in time you get it delivered. While you could let microservices interact with each other using only synchronous HTTP-based REST APIs, this might not be desirable. For instance think of downstream calls: Microservice A calls Microservice B, which then calls Microservice C and fails. How can you rollback and compensate this business transactions? Also think about availability and scalability. With synchronous communication the sender always waits for a response. If the receiver is temporarily not available, you have a problem. Also you basically block a thread on the sender and the receiver side, which obviously has some hardware-based limitations. Reliability, scalability and error handling using dead-letter queues is where asynchronous communication typically shines. Following leverage synchronous communication on the UI side of things where waiting is desired. Between the microservices use asynchronous communication. If that does not work for you, you might want to reconsider your business and service

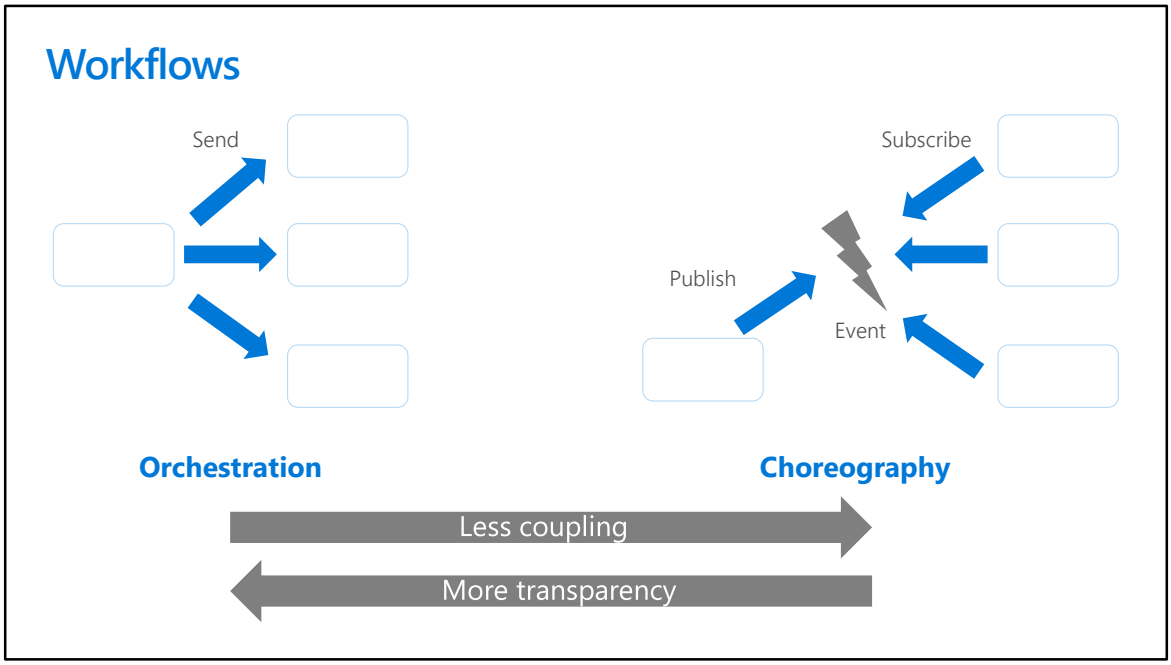
boundaries.

API gateways

- Gateway Routing
- Gateway Offloading
  - Authentication
  - SSL termination
  - IP whitelisting
  - Client throttling
  - Logging & monitoring
  - Web application firewall
- Gateway Aggregation

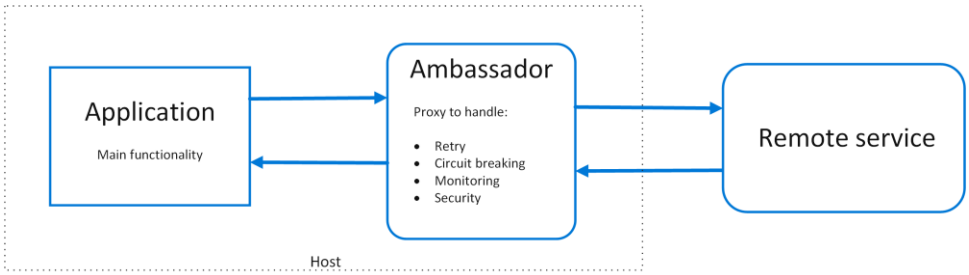


SPEAKER NOTES:
Some cross-cutting concerns are required for all microservices, i.e. authentication, SSL termination and client throttling. You do not want the management burden and let each microservice provide their own implementation of this cross-cutting concern. Better offload these concerns to a central gateway component so you can change them easily. This so-called API gateways also might help you with routing your request to the right microservice. Especially in mobile scenarios, you might want to also aggregate responses of microservices, just to take mobile network latency into consideration.



SPEAKER NOTES:
Let's talk about workflows. Workflows typically span multiple microservices, but also include a certain state to indicate its progress. You might use a dedicated workflow engine and perform orchestration, similar to the director of an orchestra. While this provides you great transparency about the progress, it also comes with a certain coupling. If less coupling is more desirable use choreography, which implicitly creates a workflow using an event-driven approach.

External systems

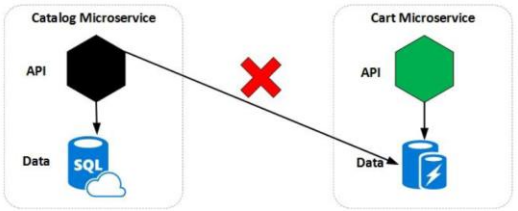


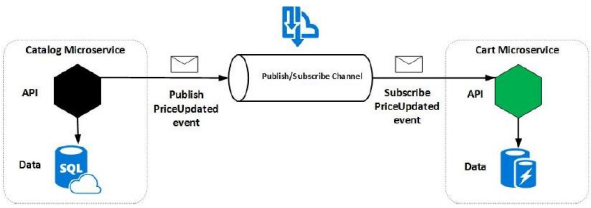
Ambassador pattern

SPEAKER NOTES:
Usually a microservice-based system does not stand for its own, it might talk to mainframes, invoke some external REST services etc. As stated, we desire asynchronous communication between the microservices to especially guarantee a certain reliability and error handling. As a solution you could plugin a proxy service in between your microservice and the remote service, the so-called Ambassador. The Ambassador provides you the capabilities the remote service is missing and also allows you to encapsulate its communication details.

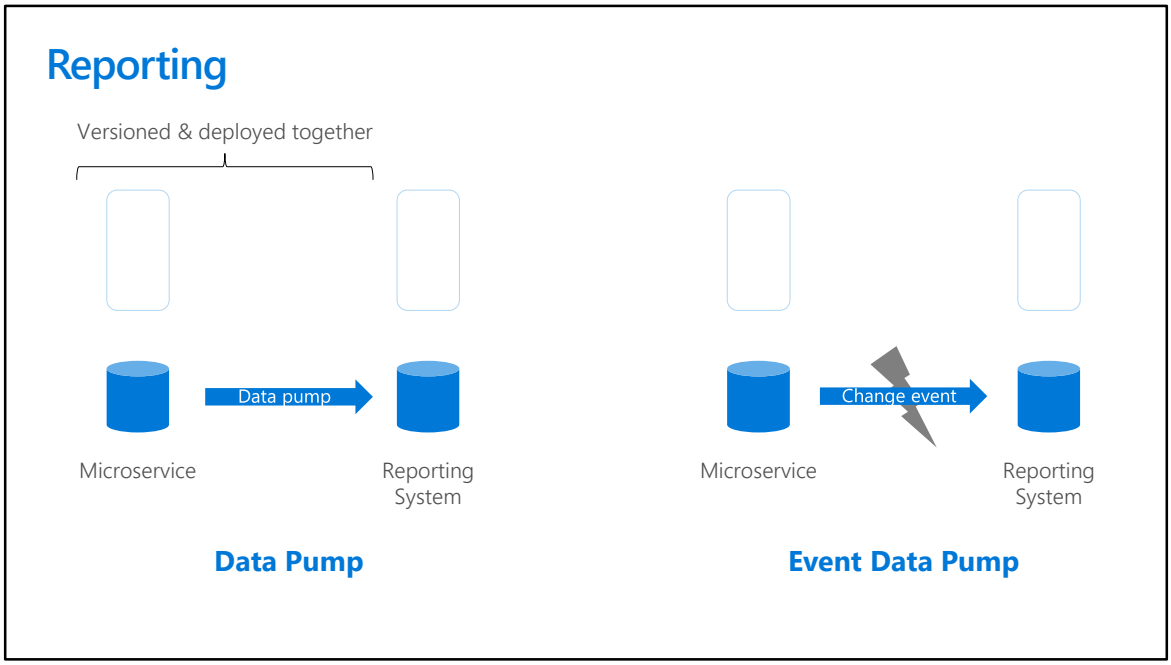
Data considerations

- ❑ Every service manages its own data
- ❑ No sharing of data stores between services
- ❑ Profit from polyglot persistence
- ❑ Service might represent the source of truth for a given entity («master»)
- ❑ Other services might hold a subset of the entity
- ❑ Embrace eventual consistency (no distributed transactions)



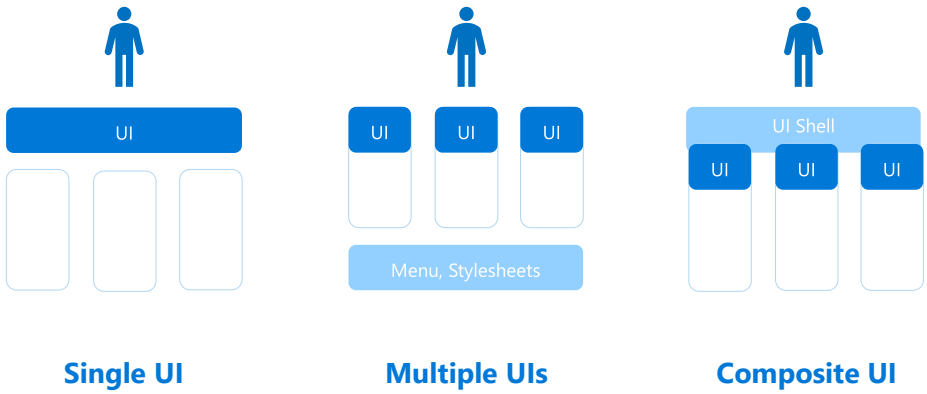


SPEAKER NOTES:
Data is one of the trickiest parts when embracing a microservice-based architecture. Every microservice manages its own data store, so there is no sharing of data stores between services. Some services might represent the source of truth for a given entity (“master”), while others might hold a subset of the entity. Takes for instance an e-commerce application. Probably there will be some user management microservice connected to a CRM system. Other microservices might also require part of the user data, i.e. the users name and shipping address are required in the shipping microservice. As stated previously, microservices should use asynchronous communication to talk to each other. So you could easily publish changes from the user management service (“master”) and subscribe from the shipping microservice. Basically embracing eventual consistency, which scales quiet well within distributed systems. Also you might embrace polyglot persistence using different data stores for different purposes, i.e. graphs, documents, etc.



SPEAKER NOTES:
As previously stated, every microservice manages its own data store, so there is no sharing of data stores between services. But there is one exception. Reporting systems need to aggregate the data in order to provide meaningful insights. Best practice here is to use data pumps, which are provided by the teams managing their microservices. When a change is required in a microservice, the corresponding team is also responsible for updating the data pump. Following the microservice and the data pump should also be versioned and deployed together. Every microservice might have a corresponding database schema in the reporting system, which itself uses materialized views for data aggregation. As stated previously, microservices should use asynchronous communication to talk to each other. Event data pumps is a flavor of data pumps, which leverage asynchronous communication to propagate change events to the reporting system. This approach is very similar to the one described for data replication in the previous slide about data considerations. Event data pumps allow even more loose coupling then traditional data pumps, but also provide less throughput. Again there is no silver bullet, you just have to decide what you want to optimize for.

UX/UI



Single UI **Multiple UIs** **Composite UI**

SPEAKER NOTES:
When splitting systems into multiple microservices there are, generally speaking, 3 choices for managing the UX/UI. One option would be to have just a single UI, which calls the microservices via REST APIs. This approach provides the greatest alignment in terms of UX/UI, but also makes the UI dependent to all microservices. You can mitigate the problem by properly versioning the REST APIs, e.g. using semantic versioning. Another approach might be that every microservices provides its own UI, resulting in multiple UIs or web apps. The menu and stylesheets would be centrally provided by another microservice. Having multiple UIs provides you with the greatest independency, but also makes alignment of UX/UI difficult. Usually it makes sense for very large UIs, which can be separated on a more coarse-grained level. A good example are large tax forms, which you develop for one tax year and unlikely change often. A mixed approach would be to develop a composite UI. Centrally only a UI Shell is provided. This UI Shell is basically a skeleton to add web components provided by different microservices. Since this web components are all embedded in the UI Shell, things like having a unified UI/UX become much easier.

DevOps

DevOps is the union of people, process, and technology to enable continuous delivery of value to customers. DevOps, a compound of dev (development) and ops (operations), is a software development practice that unifies development and IT operations. The meaning signifies coordination and collaboration among formerly siloed disciplines. Quality engineering and security teams also become part of the broader team in the DevOps model.

Continuous Integration (CI)

- Improve software development quality and speed.
- When you use Azure Pipelines or Jenkins to build apps in the cloud and deploy to Azure, each time you commit code, it's automatically built and tested, and bugs are detected faster.

Continuous Deployment (CD)

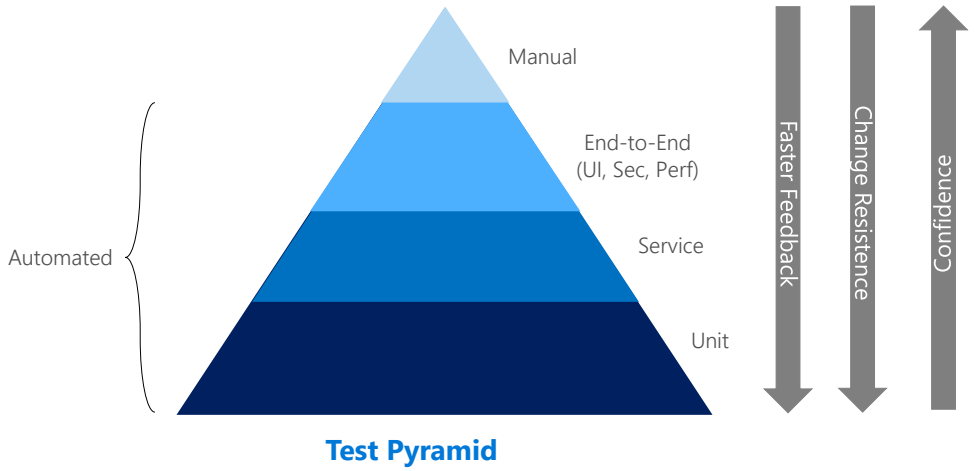
- By combining continuous integration and infrastructure as code (IaC), you'll achieve identical deployments and the confidence to deploy to production at any time.
- With continuous deployment, you can automate the entire process from code commit to production if your CI/CD tests are successful.

Continuous Learning & Monitoring

- With Azure Application Insights you can identify how your applications are performing and test if the recent deployment made things better or worse.
- Using CI/CD practices, paired with monitoring tools, you'll be able to safely deliver features to your customers as soon as they're ready.

SPEAKER NOTES:
Mastering microservices requires a high degree of automation, so DevOps is a key skill to master. Continuously integrate your code base and especially embrace automatic testing. Automatically and constantly release your microservices following the best practice of Continuous Deployment. Better have many small deployments than one large one most likely to fail. Things like testing in production may relieve you from the burden of test data management, which gets quiet complicated in every distributed system. Use continuous learning and monitoring to gain fast feedback if your changes work in production.

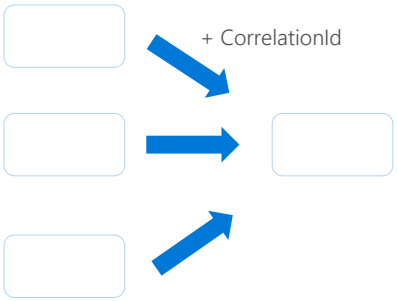
Testing



Test Pyramid

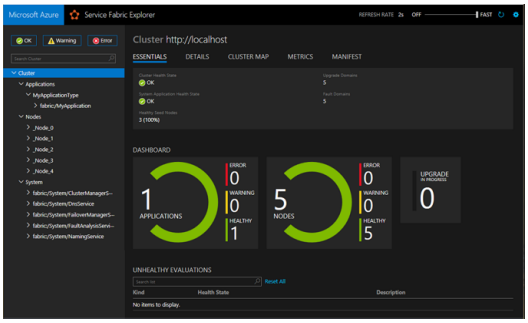
SPEAKER NOTES:
One of the more well-known models in the test automation world is the Test Pyramid. It outlines how a well-structured test suite looks like, based on the application layers being tested. While unit tests usually focus on the smaller units like classes containing business logic, service tests are testing the integration of the classes within a microservice including its data store. Some tests require a more end-to-end scope involving many microservices, like UI tests, security tests and performance tests. The key is that you automate almost everything. Manual tests should be avoided but might be required for instance if you want to do explorative manual testing first before automating via a UI test in the following sprint. Running all tests gives you the greatest confidence, but also requires a longer test duration and end-to-end tests are also more likely to break. The surface of each test category in the test pyramid also indicates the amount of tests you should have – most of them should be fast running service and even more unit tests. Since they cover already a lot of testing ground, you are already quiet confident when they succeed. Finally monitor the test coverage of your automated tests.

Continuous learning & monitoring



Microservice

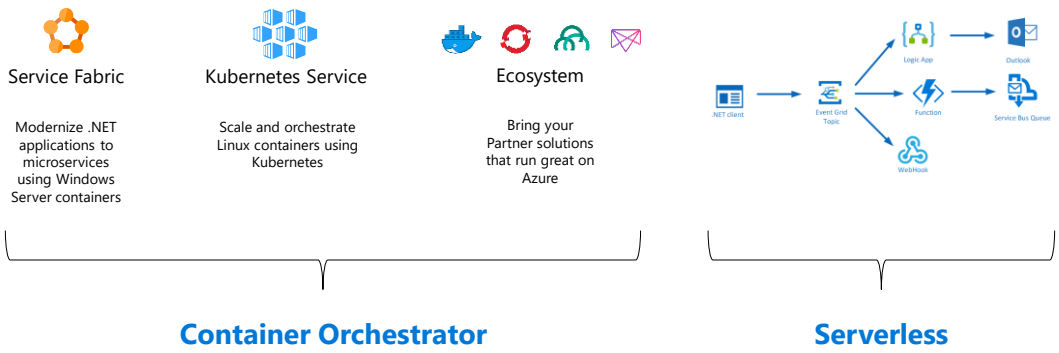
Logging System



Application monitoring
+ Drill-down into microservice

SPEAKER NOTES:
Business processes usually span multiple microservices. Since each microservice individually performs logging, a logging system is required to aggregate them. Use a CorrelationId known by all microservices to identify the instance of your business process. This helps you especially during error tracking and mitigation. Also the application monitoring should allow you drill-down from system level to each microservices to identify the root cause of an error quickly. Azure for instance provides you with Azure Monitor and the screenshot of the dashboard is actually the one provided by Service Fabric.

Hosting platform



SPEAKER NOTES:
Handling so much moving pieces in a microservice-based system, requires a proper hosting platform to orchestrate your workloads. Virtual machines require too much handcrafting and you cannot achieve high density and infrastructure utilization due to port exhaustion. Embracing containers and/or serverless is a way better solution to abstract the operational complexity. With containers, you basically package your application in a Docker container and run it on a container orchestrator like Kubernetes. This approach also helps you standardizing your DevOps approach and your hosting platform, since the container is the common denominator. They are especially helpful when modernizing existing applications in a brownfield approach, since containers can run on-premise and in the cloud. Again Azure provides you with a lot of choices here – i.e. Azure Kubernetes Service, Service Fabric, Service Fabric Mesh, RedHat OpenShift and Pivotal Cloud Foundry. When starting from greenfield, you might go even one abstraction layer higher and embrace serverless by using Azure Functions, Logic Apps and Event Grid. With serverless applications you only ship the code and do not worry about the infrastructure at all.

Microservices challenges

#1: How to identify and define the boundaries of each microservice?

- **Focus on the application's logical domain models and related data**
- **Identify decoupled islands of data and different contexts**
  - Each context could have a different business language
  - Should be defined and managed independently

Defining microservice boundaries is probably the first challenge anyone encounters. Each microservice has to be a piece of your application and each microservice should be autonomous with all the benefits and challenges that it conveys. But how do you identify those boundaries?

First, you need to focus on the application's logical domain models and related data. You must try to identify decoupled islands of data and different contexts within the same application. Each context could have a different business language (different business terms). The contexts should be defined and managed independently. The terms and entities used in those different contexts might sound similar, but you might discover that in a particular context, a business concept with one is used for a different purpose in another context and might even have a different name. For instance, a user can be referred as a user in the identity or membership context, as a customer in a CRM context, as a buyer in an ordering context, and so forth.

Microservices challenges

#2: How to create queries that retrieve data from several microservices

- Rule of thumb: avoid chatty communication
- Right solution depends on the complexity of the queries
- Always aggregate information from multiple microservices

Microservices challenges

#2: How to create queries that retrieve data from several microservices

- **Solutions:**
  - API Gateway
  - CQRS with query/reads tables
  - "Cold data" in central databases
- **If too frequent, possible indication of bad design**

The most popular solutions are the following.

API Gateway. For simple data aggregation from multiple microservices that own different databases, the recommended approach is an aggregation microservice referred to as an API Gateway. However, you need to be careful about implementing this pattern, because it can be a choke point in your system, and it can violate the principle of microservice autonomy. To mitigate this possibility, you can have multiple fined-grained API Gateways each one focusing on a vertical "slice" or business area of the system. The API Gateway pattern is explained in more detail in the section in the Using an API Gateway later.

CQRS with query/reads tables. Another solution for aggregating data from multiple microservices is the Materialized View pattern. In this approach, you generate, in advance (prepare denormalized data before the actual queries happen), a read-only table with the data that is owned

by multiple microservices. The table has a format suited to the client app's needs.
Consider something like the screen for a mobile app. If you have a single database, you might pull together the data for that screen using a SQL query that performs a complex join involving multiple tables. However, when you have multiple databases, and each database is owned by a different microservice, you cannot query those databases and create a SQL join. Your complex query becomes a challenge. You can address the requirement using a CQRS approach—you create a denormalized table in a different database that is used just for queries. The table can be designed specifically for the data you need for the complex query, with a one-to-one relationship between fields needed by your application's screen and the columns in the query table. It could also serve for reporting purposes.
This approach not only solves the original problem (how to query and join across microservices); it also improves performance considerably when compared with a complex join, because you already have the data that the application needs in the query table. Of course, using Command and Query Responsibility Segregation (CQRS) with query/reads tables means additional development work, and you will need to embrace eventual consistency. Nonetheless, requirements on performance and high scalability in collaborative scenarios (or competitive scenarios, depending on the point of view) is where you should apply CQRS with multiple databases.

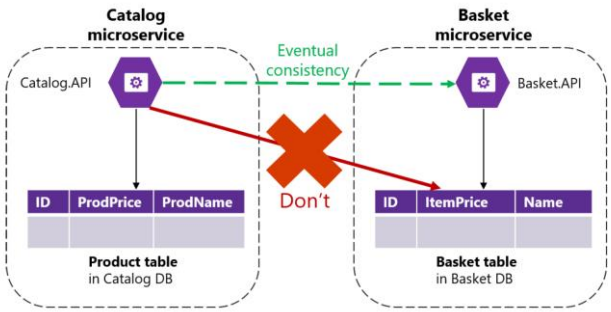
"Cold data" in central databases. For complex reports and queries that might not require real-time data, a common approach is to export your "hot data" (transactional data from the microservices) as "cold data" into large databases that are used only for reporting. That central database system can be a Big Data-based system, like Hadoop, a data warehouse like one based on Azure SQL Data Warehouse, or even a single SQL database used just for reports (if size will not be an issue).
Keep in mind that this centralized database would be used only for queries and reports that do not need real-time data. The original updates and transactions, as your source of truth, have to be in your microservices data.

However, if your application design involves constantly aggregating information from multiple microservices for complex queries, it might be a symptom of a bad design—a microservice should be as isolated as possible from other microservices. (This excludes reports/analytics that always should use cold-data central databases.) Having this problem often might be a reason to merge microservices. You need to balance the autonomy of evolution and deployment of each microservice with strong dependencies, cohesion, and data aggregation.

Microservices challenges

#3: How to achieve consistency across multiple microservices

- **Solution:**
  - use eventual consistency between microservices articulated through event-driven communication and a publish-and-subscribe system



The Catalog microservice should not update the Basket table directly, because the Basket table is owned by the Basket microservice. To make an update to the Basket microservice, the Product microservice should use eventual consistency probably based on asynchronous communication such as integration events (message and event-based communication).

Moreover, ACID-style or two-phase commit transactions are not just against microservices principles; most NoSQL databases (like Azure Cosmos DB, MongoDB, etc.) do not support two-phase commit transactions, typical in distributed databases scenarios. However, maintaining data consistency across services and databases is essential. This challenge is also related to the question of how to propagate changes across multiple microservices when certain data needs to be redundant—for example, when you need to have the product's name or description in the Catalog microservice and the Basket microservice.

Therefore, as a conclusion, a good solution for this problem is to use eventual consistency between microservices articulated through event-driven communication and a publish-and-subscribe system.

Microservices challenges

#4: How to design communication across microservice boundaries

- **Rely on HTTP REST calls for client-service communication**
- **Don't create long chains of calls between microservices**
  - Blocking and low performance
  - Coupling microservices with HTTP
  - Failure in any one microservice

A popular approach is to implement HTTP (REST)- based microservices, due to their simplicity. An HTTP-based approach is perfectly acceptable; the issue here is related to how you use it. If you use HTTP requests and responses just to interact with your microservices from client applications or from API Gateways, that is fine. But if you create long chains of synchronous HTTP calls across microservices, communicating across their boundaries as if the microservices were objects in a monolithic application, your application will eventually run into problems.

For instance, imagine that your client application makes an HTTP API call to an individual microservice like the Ordering microservice. If the Ordering microservice in turn calls additional microservices using HTTP within the same request/response cycle, you are creating a chain of HTTP calls. It might sound reasonable initially. However, there are important points to consider when going down this path:

• Blocking and low performance. Due to the synchronous nature of HTTP,

the original request will not get a response until all the internal HTTP calls are finished. Imagine if the number of these calls increases significantly and at the same time one of the intermediate HTTP calls to a microservice is blocked. The result is that performance is impacted, and the overall scalability will be exponentially affected as additional HTTP requests increase.

• Coupling microservices with HTTP. Business microservices should not be coupled with other business microservices. Ideally, they should not "know" about the existence of other microservices. If your application relies on coupling microservices as in the example, achieving autonomy per microservice will be almost impossible.

• Failure in any one microservice. If you implemented a chain of microservices linked by HTTP calls, when any of the microservices fails (and eventually they will fail) the whole chain of microservices will fail. A microservice-based system should be designed to continue to work as well as possible during partial failures. Even if you implement client logic that uses retries with exponential backoff or circuit breaker mechanisms, the more complex the HTTP call chains are, the more complex it is implement a failure strategy based on HTTP.

Microservices challenges

#4: How to design communication across microservice boundaries

- **HTTP chaining arguably means monolithic application with HTTP calls instead of intraprocess communication calls**
- **Solution:**
  - Minimize HTTP chaining
  - Rely on asynchronous interaction for inter-microservice comm
  - Use event-based communication or async HTTP polling

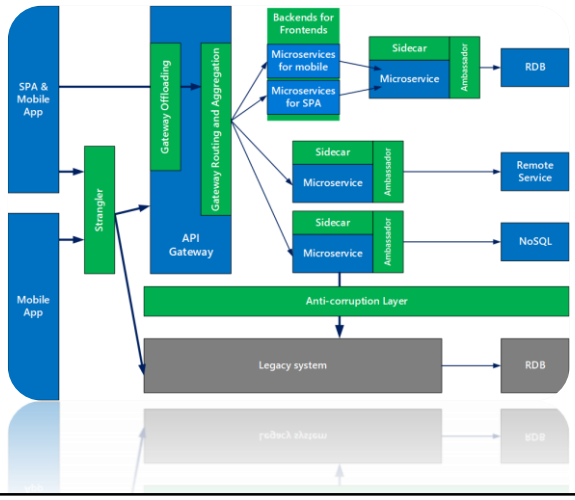
In fact, if your internal microservices are communicating by creating chains of HTTP requests as described, it could be argued that you have a monolithic application, but one based on HTTP between processes instead of intraprocess communication mechanisms.

Therefore, in order to enforce microservice autonomy and have better resiliency, you should minimize the use of chains of request/response communication across microservices. It is recommended that you use only asynchronous interaction for inter-microservice communication, either by using asynchronous message- and event-based communication, or by using (asynchronous) HTTP polling independently of the original HTTP request/response cycle.

Ramp-up

Consider microservices for:

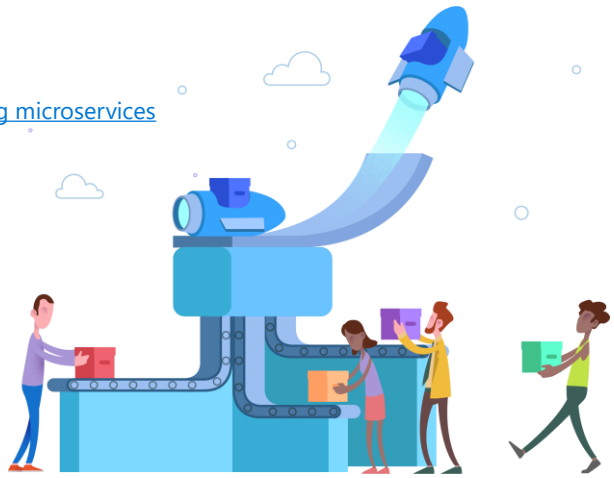
- Apps with many subdomains
- Scaling towards multiple teams
- Require high release velocity
- Require granular scalability



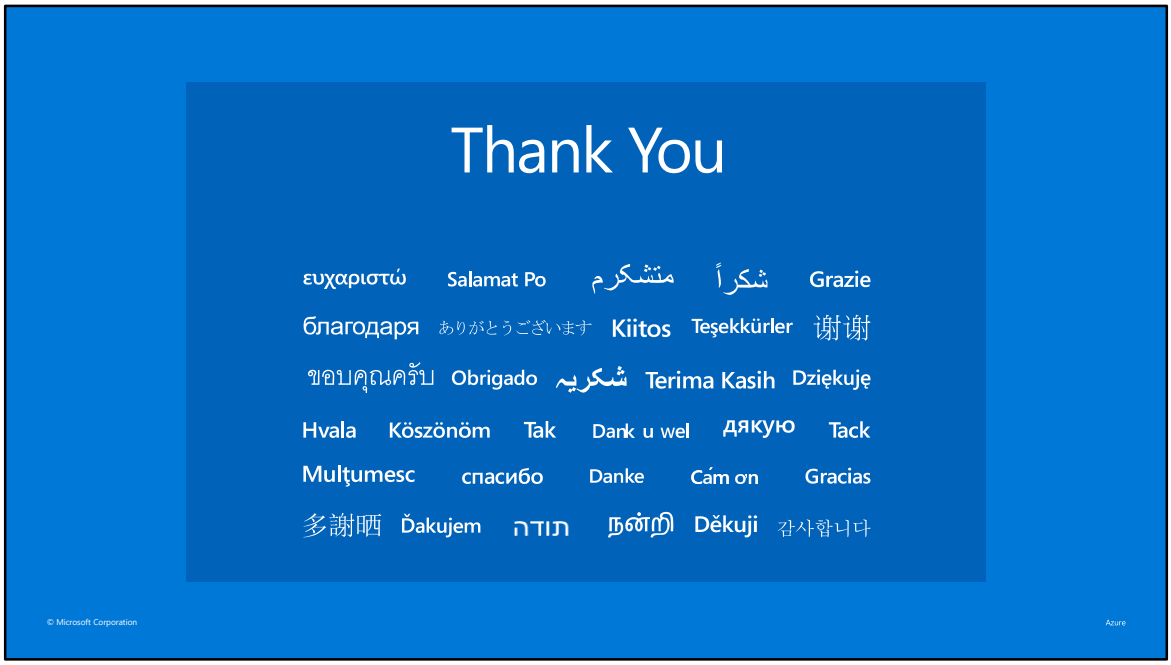
SPEAKER NOTES:
As we have seen, microservices are no silver bullet. Microservices are solving organizational problems by introducing technical challenges. They follow a rather holistic approach so they can evolve individually, basically by aligning business, services and team boundaries. Following they are great for apps with many subdomains, when scaling towards multiple teams, if you require a higher release velocity or a more granular scalability. In order to compensate the technical challenges coming with it, you might rely on a mature and well-engineered platform like Microsoft Azure.

Additional resources

- Microservices architecture style
- Designing, building, and operating microservices
- Design patterns for microservices
- Microservice Guide
- Microservices



SPEAKER NOTES:
Finally, take a look at this links to further deep-dive into microservices. Also, look into related topics like DevOps, Containers and especially Azure. Happy coding.



What are Microservices?

What are Microservices?

Microservices

From Wikipedia, the free encyclopedia

Some of this article's listed sources **may not be reliable**. Please help this article by looking for better, more reliable sources. Unreliable citations may be challenged or deleted. *(October 2018) (Learn how and when to remove this template message)*

Microservices are a software development technique—a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. In a microservices architecture, services are fine-grained and the protocols are lightweight. The benefit of decomposing an application into different smaller services is that it improves modularity. This makes the application easier to understand, develop, test, and become more resilient to architecture erosion.[1] It parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently.[2] It also allows the architecture of an individual service to emerge through continuous refactoring.[3] Microservices-based architectures enable continuous delivery and deployment.[4]

Avælgo

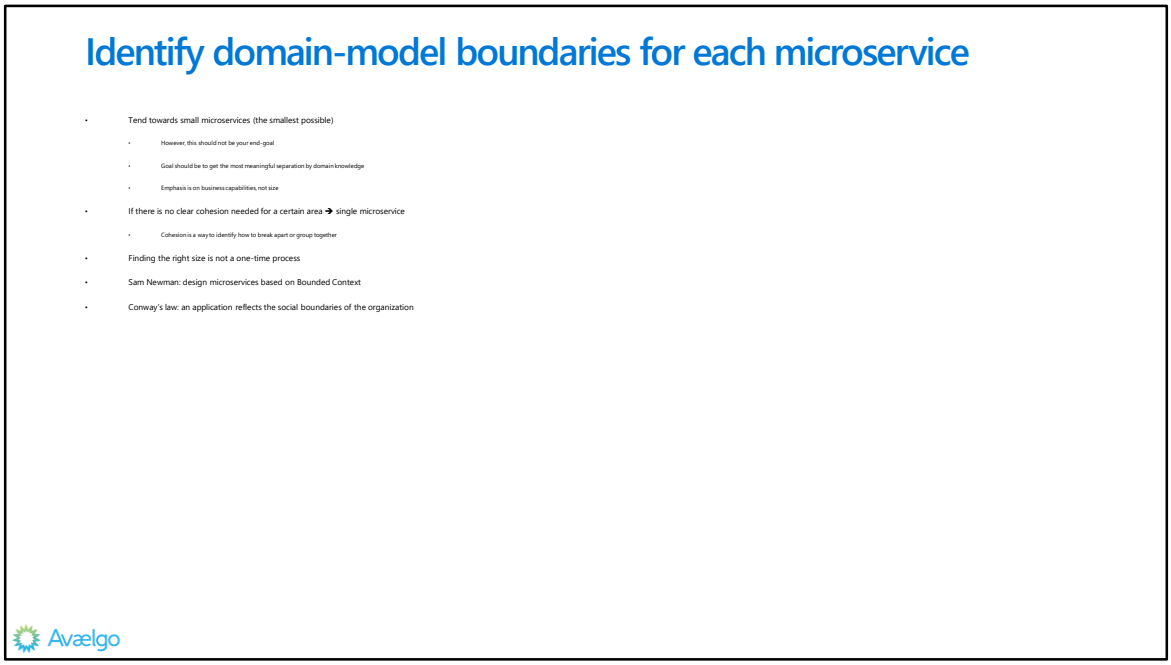
Challenges and solutions for distributed data management

Challenge #1: How to define the boundaries of each microservice

- Boundaries = first challenge
- Each microservice has a piece of the application
- Each microservice should be autonomous

- Focus on the logical domain and related data
- Try to identify decoupled islands of data and different contexts within the application
- Each context = different business term
- Contexts should be defined and managed independently
  - e. g. a user → user in identity, customer in CRM, buyer in ordering etc.

Avælgo



The goal when identifying model boundaries and size for each microservice isn't to get to the most granular separation possible, although you should tend toward small microservices if possible. Instead, your goal should be to get to the most meaningful separation guided by your domain knowledge. The emphasis isn't on the size, but instead on business capabilities. In addition, if there's clear cohesion needed for a certain area of the application based on a high number of dependencies, that indicates the need for a single microservice, too. Cohesion is a way to identify how to break apart or group together microservices. Ultimately, while you gain more knowledge about the domain, you should adapt the size of your microservice, iteratively. Finding the right size isn't a one-shot process.

Sam Newman, a recognized promoter of microservices and author of the book Building Microservices, highlights that you should design your microservices based on the Bounded Context (BC) pattern (part of domain-driven design), as introduced earlier. Sometimes, a BC could be composed of several physical services, but not vice versa.

A domain model with specific domain entities applies within a concrete BC or microservice. A BC delimits the applicability of a domain model and gives developer

team members a clear and shared understanding of what must be cohesive and what can be developed independently. These are the same goals for microservices.

Another tool that informs your design choice is Conway's law, which states that an application will reflect the social boundaries of the organization that produced it. But sometimes the opposite is true -the company's organization is formed by the software. You might need to reverse Conway's law and build the boundaries the way you want the company to be organized, leaning toward business process consulting.

Identify domain-model boundaries for each microservice

- Context Mapping: identify various contexts in the application and their boundaries
- In larger applications, domain models will quickly get fragmented
  - Domain experts will name entities differently
  - Domain experts choose to store or ditch attributes based on requirements
- Don't try to unify the teams
  - Accept richness provided by each domain
  - Otherwise, a central database will end up with awkward domain attributes, sounding fishy for every single domain expert

Avælgo

To identify bounded contexts, you can use a DDD pattern called the Context Mapping pattern. With Context Mapping, you identify the various contexts in the application and their boundaries. It's common to have a different context and boundary for each small subsystem, for instance. The Context Map is a way to define and make explicit those boundaries between domains. A BC is autonomous and includes the details of a single domain -details like the domain entities- and defines integration contracts with other BCs. This is similar to the definition of a microservice: it's autonomous, it implements certain domain capability, and it must provide interfaces. This is why Context Mapping and the Bounded Context pattern are good approaches for identifying the domain model boundaries of your microservices.

When designing a large application, you'll see how its domain model can be fragmented - a domain expert from the catalog domain will name entities differently in the catalog and inventory domains than a shipping domain expert, for instance. Or the user domain entity might be different in size and number of attributes when dealing with a CRM expert who wants to store every detail about the customer than for an ordering domain expert who just needs partial data about the customer. It's very hard to disambiguate all domain terms across all the domains related to a large application. But the most important thing is that you shouldn't try to unify the terms. Instead,

accept the differences and richness provided by each domain. If you try to have a unified database for the whole application, attempts at a unified vocabulary will be awkward and won't sound right to any of the multiple domain experts. Therefore, BCs (implemented as microservices) will help you to clarify where you can use certain domain terms and where you'll need to split the system and create additional BCs with different domains.

Identify domain-model boundaries for each microservice

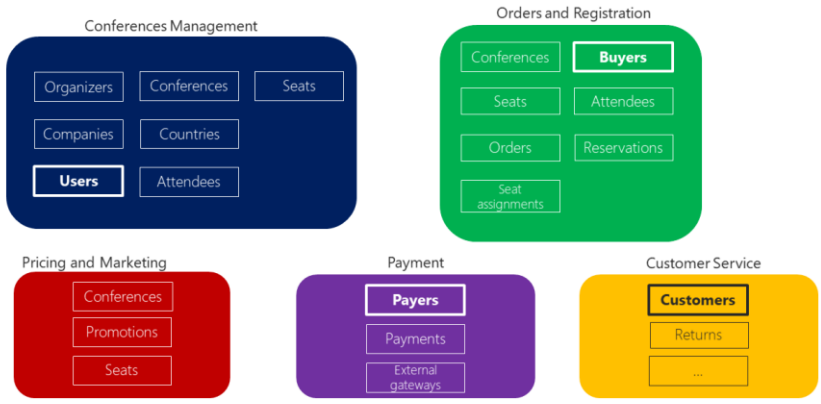
- An indication of a properly designed boundary and size for each business context is when:
  - There are few strong relationships between domain models
  - You don't need to merge information from multiple domain models
- Sizing best answer:
  - It should be an autonomous BS
  - It should be as isolated as possible
  - It should allow you to work without having to switch to other contexts

Avælgo

You'll know that you got the right boundaries and sizes of each BC and domain model if you have few strong relationships between domain models, and you do not usually need to merge information from multiple domain models when performing typical application operations.

Perhaps the best answer to the question of how large a domain model for each microservice should be is the following: it should have an autonomous BC, as isolated as possible, that enables you to work without having to constantly switch to other contexts (other microservice's models). In Figure 4-10, you can see how multiple microservices (multiple BCs) each has their own model and how their entities can be defined, depending on the specific requirements for each of the identified domains in your application.

Identify domain-model boundaries for each microservice



Avaelgo

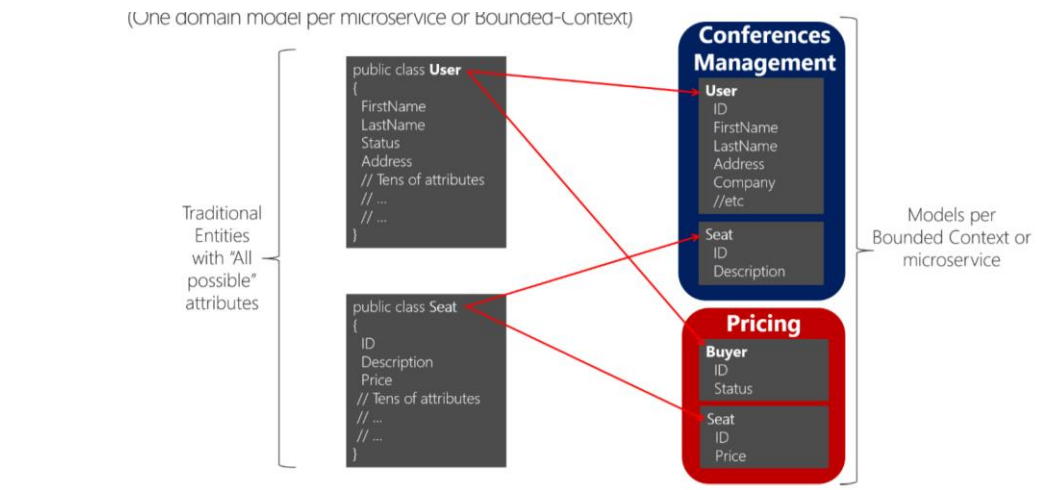
You can see how multiple microservices (multiple BCs) each has their own model and how their entities can be defined, depending on the specific requirements for each of the identified domains in your application

This figure illustrates a sample scenario related to an online conference management system. You've identified several BCs that could be implemented as microservices, based on domains that domain experts defined for you. As you can see, there are entities that are present just in a single microservice model, like Payments in the Payment microservice. Those will be easy to implement.

However, you might also have entities that have a different shape but share the same identity across the multiple domain models from the multiple microservices. For example, the User entity is identified in the Conferences Management microservice. That same user, with the same identity, is the one named Buyers in the Ordering microservice, or the one named Payer in the Payment microservice, and even the one named Customer in the Customer Service microservice. This is because, depending on the ubiquitous language that each domain expert is using, a user might have a different perspective even with different attributes. The user entity in the microservice model named Conferences Management might have most of its personal data

attributes. However, that same user in the shape of Payer in the microservice Payment or in the shape of Customer in the microservice Customer Service might not need the same list of attributes.

Identify domain-model boundaries for each microservice



Avaelgo

You can see how the user is present in the Conferences Management microservice model as the User entity and is also present in the form of the Buyer entity in the Pricing microservice, with alternate attributes or details about the user when it's actually a buyer. Each microservice or BC might not need all the data related to a User entity, just part of it, depending on the problem to solve or the context. For instance, in the Pricing microservice model, you do not need the address or the name of the user, just the ID (as identity) and Status, which will have an impact on discounts when pricing the seats per buyer.

The Seat entity has the same name but different attributes in each domain model. However, Seat shares identity based on the same ID, as happens with User and Buyer.

Identify domain-model boundaries for each microservice

- Benefits of not sharing the same entity with the same number of attributes across domains:
  - Reduce duplication
  - A master microservice: updates and queries for a type of data are driven only by a microservice, without leaving "options"

There are several benefits to not sharing the same user entity with the same number of attributes across domains. One benefit is to reduce duplication, so that microservice models do not have any data that they do not need. Another benefit is having a master microservice that owns a certain type of data per entity so that updates and queries for that type of data are driven only by that microservice.

Challenge #2: How to create queries that retrieve data from several microservices

- Cross-service querying is likely the second challenge
- Need to avoid chatty communication and communication chaining
  - e. g. a mobile app screen that shows the user information, product catalog, basket information etc.
- API Gateway: a simple data aggregation from multiple microservices
- Careful about API gateways → tend to be a choke point and to violate the autonomy principal

Avælgo

A second challenge is how to implement queries that retrieve data from several microservices, while avoiding chatty communication to the microservices from remote client apps. An example could be a single screen from a mobile app that needs to show user information that's owned by the basket, catalog, and user identity microservices. Another example would be a complex report involving many tables located in multiple microservices. The right solution depends on the complexity of the queries. But in any case, you'll need a way to aggregate information if you want to improve the efficiency in the communications of your system. The most popular solutions are the following.

API Gateway. For simple data aggregation from multiple microservices that own different databases, the recommended approach is an aggregation microservice referred to as an API Gateway. However, you need to be careful about implementing this pattern, because it can be a choke point in your system, and it can violate the principle of microservice autonomy. To mitigate this possibility, you can have multiple fined-grained API Gateways each one focusing on a vertical "slice" or business area of the system.

Challenge #2: How to create queries that retrieve data from several microservices

- Cross-service querying is likely the second challenge
- Need to avoid chatty communication and communication chaining
  - e. g. a mobile app screen that shows the user information, product catalog, basket information etc.
- Command and Query segregation responsibility: a Materialized View where some denormalized data is prepared in advance as a read-only table with the data owned by multiple microservices

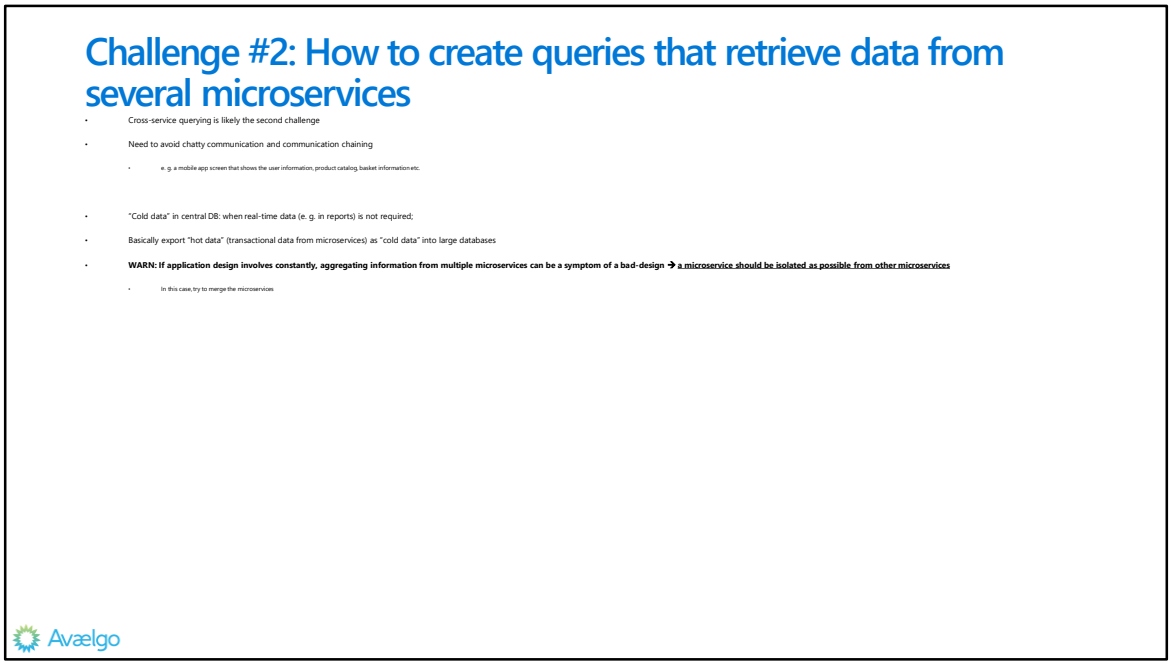
Avælgo

CQRS with query/reads tables. Another solution for aggregating data from multiple microservices is the Materialized View pattern. In this approach, you generate, in advance (prepare denormalized data before the actual queries happen), a read-only table with the data that's owned by multiple microservices. The table has a format suited to the client app's needs.

Consider something like the screen for a mobile app. If you have a single database, you might pull together the data for that screen using a SQL query that performs a complex join involving multiple tables. However, when you have multiple databases, and each database is owned by a different microservice, you cannot query those databases and create a SQL join. Your complex query becomes a challenge. You can address the requirement using a CQRS approach—you create a denormalized table in a different database that's used just for queries. The table can be designed specifically for the data you need for the complex query, with a one-to-one relationship between fields needed by your application's screen and the columns in the query table. It could also serve for reporting purposes.

This approach not only solves the original problem (how to query and join across microservices), but it also improves performance considerably when compared with a complex join, because you already have the data that the application needs in the query table. Of course, using Command and Query Responsibility Segregation (CQRS)

with query/reads tables means additional development work, and you'll need to embrace eventual consistency. Nonetheless, requirements on performance and high scalability in collaborative scenarios (or competitive scenarios, depending on the point of view) are where you should apply CQRS with multiple databases.



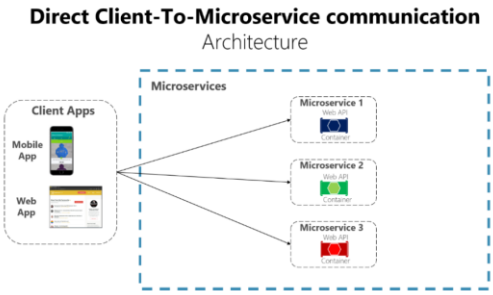
"Cold data" in central databases. For complex reports and queries that might not require real-time data, a common approach is to export your "hot data" (transactional data from the microservices) as "cold data" into large databases that are used only for reporting. That central database system can be a Big Data-based system, like Hadoop, a data warehouse like one based on Azure SQL Data Warehouse, or even a single SQL database that's used just for reports (if size won't be an issue).
Keep in mind that this centralized database would be used only for queries and reports that do not need real-time data. The original updates and transactions, as your source of truth, have to be in your microservices data. The way you would synchronize data would be either by using event-driven communication (covered in the next sections) or by using other database infrastructure import/export tools. If you use event-driven communication, that integration process would be similar to the way you propagate data as described earlier for CQRS query tables.
However, if your application design involves constantly aggregating information from multiple microservices for complex queries, it might be a symptom of a bad design -a microservice should be as isolated as possible from other microservices. (This excludes reports/analytics that always should use cold-data central databases.) Having this problem often might be a reason to merge microservices. You need to balance the autonomy of evolution and deployment of each microservice with strong

dependencies, cohesion, and data aggregation.

API Gateway Pattern vs. the Direct client-to-microservice communication

- Each microservice (should) exposes a set of (typically) fine-grained endpoints

Direct client-to-microservice communication



Avaelgo

In a microservices architecture, each microservice exposes a set of (typically) fine-grained endpoints. This fact can impact the client-to-microservice communication, as explained in this section.

Direct client-to-microservice communication
A possible approach is to use a direct client-to-microservice communication architecture. In this approach, a client app can make requests directly to some of the microservices, as shown in the figure.
In this approach, each microservice has a public endpoint, sometimes with a different TCP port for each microservice.

An example of a URL for a particular service could be the following URL in Azure: http://eshoponcontainers.westus.cloudapp.azure.com:88/

In a production environment based on a cluster, that URL would map to the load balancer used in the cluster, which in turn distributes the requests across the microservices. In production environments, you could have an Application Delivery Controller (ADC) like Azure Application Gateway between your microservices and the Internet. This acts as a transparent tier that not only performs load balancing, but

secures your services by offering SSL termination. This improves the load of your hosts by offloading CPU-intensive SSL termination and other routing duties to the Azure Application Gateway. In any case, a load balancer and ADC are transparent from a logical application architecture point of view.

A direct client-to-microservice communication architecture could be good enough for a small microservice-based application, especially if the client app is a server-side web application like an ASP.NET MVC app. However, when you build large and complex microservice-based applications (for example, when handling dozens of microservice types), and especially when the client apps are remote mobile apps or SPA web applications, that approach faces a few issues

API Gateway Pattern vs. the Direct client-to-microservice communication

- Questions to consider when developing large(r) microservices-based applications
  - How can client apps minimize the number of requests to the back end and reduce chatty communication to multiple microservices?
  - How can you handle cross-cutting concerns such as authorization, data transformations and dynamic request dispatching?
  - How can client apps communicate with services that use non-Internet-friendly protocols?
  - How can you shape a façade especially made for mobile apps?

Avælgo

How can client apps minimize the number of requests to the back end and reduce chatty communication to multiple microservices?

- Interacting with multiple microservices to build a single UI screen increases the number of round trips across the Internet. This increases latency and complexity on the UI side. Ideally, responses should be efficiently aggregated in the server side. This reduces latency, since multiple pieces of data come back in parallel and some UI can show data as soon as it's ready.

How can you handle cross-cutting concerns such as authorization, data transformations, and dynamic request dispatching?

- Implementing security and cross-cutting concerns like security and authorization on every microservice can require significant development effort. A possible approach is to have those services within the Docker host or internal cluster to restrict direct access to them from the outside, and to implement those cross-cutting concerns in a centralized place, like an API Gateway.

*How can client apps communicate with services that use non-Internet-friendly protocols?**

- Protocols used on the server side (like AMQP or binary protocols) are usually not

supported in client apps. Therefore, requests must be performed through protocols like HTTP/HTTPS and translated to the other protocols afterwards. A *man-in-the-middle* approach can help in this situation.

How can you shape a facade especially made for mobile apps?

- Protocols used on the server side (like AMQP or binary protocols) are usually not supported in client apps. Therefore, requests must be performed through protocols like HTTP/HTTPS and translated to the other protocols afterwards. A *man-in-the-middle* approach can help in this situation.

API Gateway Pattern vs. the Direct client-to-microservice communication

Why consider API Gateways instead?

- No more coupling between client apps and microservices
- Less overall round-trips
- Better security by having less attach surface areas
- Less boiler-plate code regarding cross-cutting concerns

Avælgo

In a microservices architecture, the client apps usually need to consume functionality from more than one microservice. If that consumption is performed directly, the client needs to handle multiple calls to microservice endpoints. What happens when the application evolves and new microservices are introduced or existing microservices are updated? If your application has many microservices, handling so many endpoints from the client apps can be a nightmare. Since the client app would be coupled to those internal endpoints, evolving the microservices in the future can cause high impact for the client apps.

Therefore, having an intermediate level or tier of indirection (Gateway) can be very convenient for microservice-based applications. If you don't have API Gateways, the client apps must send requests directly to the microservices and that raises problems, such as the following issues:

- Coupling: Without the API Gateway pattern, the client apps are coupled to the internal microservices. The client apps need to know how the multiple areas of the application are decomposed in microservices. When evolving and refactoring the internal microservices, those actions impact maintenance pretty badly because they cause breaking changes to the client apps due to the direct reference to the internal microservices from the client apps. Client apps need to be updated

frequently, making the solution harder to evolve.

- **Too many round trips**: A single page/screen in the client app might require several calls to multiple services. That can result in multiple network round trips between the client and the server, adding significant latency. Aggregation handled in an intermediate level could improve the performance and user experience for the client app.
- **Security issues**: Without a gateway, all the microservices must be exposed to the "external world", making the attack surface larger than if you hide internal microservices that aren't directly used by the client apps. The smaller the attack surface is, the more secure your application can be.
- **Cross-cutting concerns**: Each publicly published microservice must handle concerns such as authorization, SSL, etc. In many situations, those concerns could be handled in a single tier so the internal microservices are simplified.

API Gateway Pattern vs. the Direct client-to-microservice communication

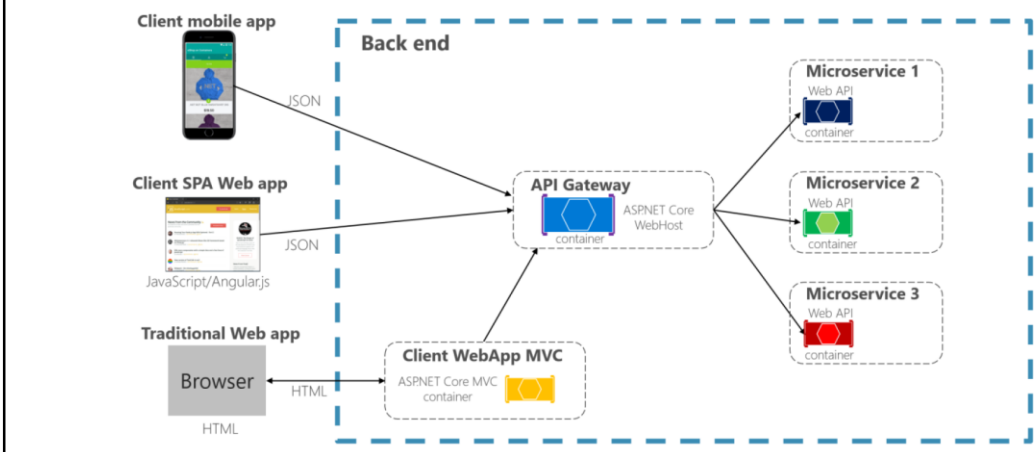
What is the API Gateway?

- A service that provides a single entry point for certain groups of microservices
- Similar to the Façade pattern from OOP
- Also known as BFF (backend for frontend)
- Sits between client apps and the microservice
- Acts as a reverse-proxy, routing requests from clients to services
- Optionally provides cross-cutting concerns, like authentication, SSL termination and caching

When you design and build large or complex microservice-based applications with multiple client apps, a good approach to consider can be an API Gateway. This is a service that provides a single-entry point for certain groups of microservices. It's similar to the Facade pattern from object-oriented design, but in this case, it's part of a distributed system. The API Gateway pattern is also sometimes known as the "backend for frontend" (BFF) because you build it while thinking about the needs of the client app.

Therefore, the API gateway sits between the client apps and the microservices. It acts as a reverse proxy, routing requests from clients to services. It can also provide additional cross-cutting features such as authentication, SSL termination, and cache.

API Gateway Pattern vs. the Direct client-to-microservice communication



Avaelgo

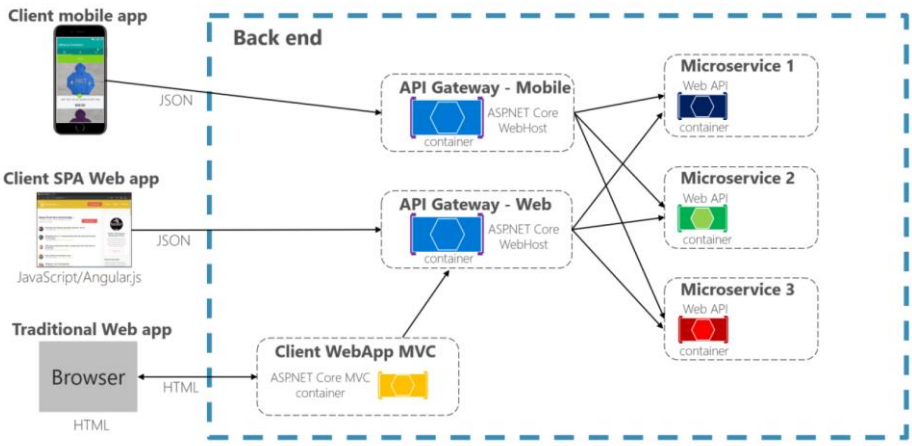
In this example, the API Gateway would be implemented as a custom ASP.NET Core WebHost service running as a container.

It's important to highlight that in that diagram, you would be using a single custom API Gateway service facing multiple and different client apps. That fact can be an important risk because your API Gateway service will be growing and evolving based on many different requirements from the client apps. Eventually, it will be bloated because of those different needs and effectively it could be pretty similar to a monolithic application or monolithic service. That's why it's very much recommended to split the API Gateway in multiple services or multiple smaller API Gateways, one per client app form-factor type, for instance.

You need to be careful when implementing the API Gateway pattern. Usually it isn't a good idea to have a single API Gateway aggregating all the internal microservices of your application. If it does, it acts as a monolithic aggregator or orchestrator and violates microservice autonomy by coupling all the microservices.

Therefore, the API Gateways should be segregated based on business boundaries and the client apps and not act as a single aggregator for all the internal microservices.

API Gateway Pattern vs. the Direct client-to-microservice communication



Avaelgo

When splitting the API Gateway tier into multiple API Gateways, if your application has multiple client apps, that can be a primary pivot when identifying the multiple API Gateways types, so that you can have a different facade for the needs of each client app. This case is a pattern named “Backend for Frontend” (BFF) where each API Gateway can provide a different API tailored for each client app type, possibly even based on the client form factor by implementing specific adapter code which underneath calls multiple internal microservices, as shown in the image.

API Gateway Pattern vs. the Direct client-to-microservice communication

API Gateway features:

- Reverse proxy or gateway routing
- Request aggregation
- Cross-cutting concerns or gateway offloading: authentication, authorization, service discovery integration, response caching, retry policies, circuit breaker and QoS, rate limiting and throttling, load balancing, logging, tracing, correlation, headers, query strings and claims transformation, IP whitelisting

An API Gateway can offer multiple features. Depending on the product it might offer richer or simpler features, however, the most important and foundational features for any API Gateway are the following design patterns:

Reverse proxy or gateway routing. The API Gateway offers a reverse proxy to redirect or route requests (layer 7 routing, usually HTTP requests) to the endpoints of the internal microservices. The gateway provides a single endpoint or URL for the client apps and then internally maps the requests to a group of internal microservices. This routing feature helps to decouple the client apps from the microservices but it's also pretty convenient when modernizing a monolithic API by sitting the API Gateway in between the monolithic API and the client apps, then you can add new APIs as new microservices while still using the legacy monolithic API until it's split into many microservices in the future. Because of the API Gateway, the client apps won't notice if the APIs being used are implemented as internal microservices or a monolithic API and more importantly, when evolving and refactoring the monolithic API into microservices, thanks to the API Gateway routing, client apps won't be impacted with any URI change.

Requests aggregation. As part of the gateway pattern you can aggregate multiple

client requests (usually HTTP requests) targeting multiple internal microservices into a single client request. This pattern is especially convenient when a client page/screen needs information from several microservices. With this approach, the client app sends a single request to the API Gateway that dispatches several requests to the internal microservices and then aggregates the results and sends everything back to the client app. The main benefit and goal of this design pattern is to reduce chattiness between the client apps and the backend API, which is especially important for remote apps out of the datacenter where the microservices live, like mobile apps or requests coming from SPA apps that come from Javascript in client remote browsers. For regular web apps performing the requests in the server environment (like an ASP.NET Core MVC web app), this pattern is not so important as the latency is very much smaller than for remote client apps.

Cross-cutting concerns or gateway offloading. Depending on the features offered by each API Gateway product, you can offload functionality from individual microservices to the gateway, which simplifies the implementation of each microservice by consolidating cross-cutting concerns into one tier. This is especially convenient for specialized features that can be complex to implement properly in every internal microservice, such as the following functionality:

- Authentication and authorization
- Service discovery integration
- Response caching
- Retry policies, circuit breaker, and QoS
- Rate limiting and throttling
- Load balancing
- Logging, tracing, correlation
- Headers, query strings, and claims transformation
- IP whitelisting

API Gateway Pattern vs. the Direct client-to-microservice communication

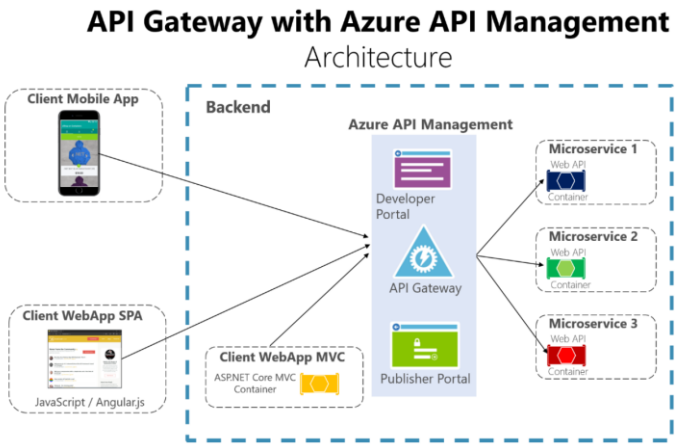
Using products with API Gateway features:

- Azure API Management
- Ocelot

Avælgo

API Gateway Pattern vs. the Direct client-to-microservice communication

Azure API Management



Avaelgo

In this case, when using a product like Azure API Management, the fact that you might have a single API Gateway is not so risky because these kinds of API Gateways are "thinner", meaning that you don't implement custom C# code that could evolve towards a monolithic component.

The API Gateway products usually act like a reverse proxy for ingress communication, where you can also filter the APIs from the internal microservices plus apply authorization to the published APIs in this single tier.
The insights available from an API Management system help you get an understanding of how your APIs are being used and how they are performing. They do this by letting you view near real-time analytics reports and identifying trends that might impact your business. Plus, you can have logs about request and response activity for further online and offline analysis.

With Azure API Management, you can secure your APIs using a key, a token, and IP filtering. These features let you enforce flexible and fine-grained quotas and rate limits, modify the shape and behavior of your APIs using policies, and improve performance with response caching.

API Gateway Pattern vs. the Direct client-to-microservice communication

Ocelot

- Lightweight API Gateway
- Open Source .NET Core based API Gateway especially made for microservices
- Previous diagrams represent Ocelot
- Alternatives: Apigee, Kong, MuleSoft, WSO2, Linkerd, Istio

Avælgo

Ocelot is a lightweight API Gateway, recommended for simpler approaches. Ocelot is an Open Source .NET Core based API Gateway especially made for microservices architecture that need unified points of entry into their system. It's lightweight, fast, scalable and provides routing and authentication among many other features.
The main reason to choose Ocelot for the eShopOnContainers reference application is because Ocelot is a .NET Core lightweight API Gateway that you can deploy into the same application deployment environment where you're deploying your microservices/containers, such as a Docker Host, Kubernetes, Service Fabric, etc. And since it's based on .NET Core, it's cross-platform allowing you to deploy on Linux or Windows.
The previous diagrams showing custom API Gateways running in containers are precisely how you can also run Ocelot in a container and microservice-based application.
In addition, there are many other products in the market offering API Gateways features, such as Apigee, Kong, MuleSoft, WSO2, and other products like Linkerd and Istio for service mesh ingress controller features.

API Gateway Pattern vs. the Direct client-to-microservice communication

Drawbacks of the API Gateway pattern

- Strong coupling
- Single point of failure
- Can introduce increased response time due to additional network call
- Can become a bottleneck under high load, if improperly scaled
- Requires additional development cost
- If developed by a single team, development can be a bottlenect

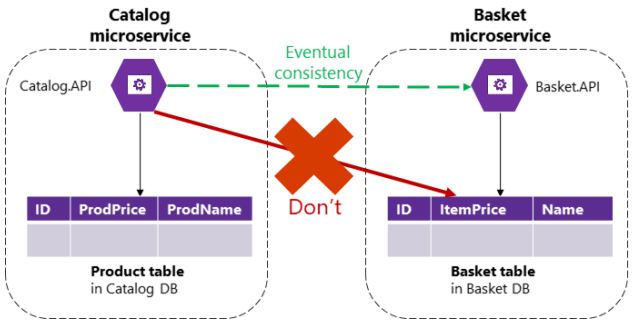
Avælgo

• The most important drawback is that when you implement an API Gateway, you're coupling that tier with the internal microservices. Coupling like this might introduce serious difficulties for your application. Clemens Vaster, architect at the Azure Service Bus team, refers to this potential difficulty as "the new ESB" in the "Messaging and Microservices" session at GOTO 2016.
• Using a microservices API Gateway creates an additional possible single point of failure.
• An API Gateway can introduce increased response time due to the additional network call. However, this extra call usually has less impact than having a client interface that's too chatty directly calling the internal microservices.
• If not scaled out properly, the API Gateway can become a bottleneck.
• An API Gateway requires additional development cost and future maintenance if it includes custom logic and data aggregation. Developers must update the API Gateway in order to expose each microservice's endpoints. Moreover, implementation changes in the internal microservices might cause code changes at the API Gateway level. However, if the API Gateway is just applying security, logging, and versioning (as when using Azure API Management), this additional development cost might not apply.
• If the API Gateway is developed by a single team, there can be a development bottleneck. This is another reason why a better approach is to have several fined-

grained API Gateways that respond to different client needs. You could also segregate the API Gateway internally into multiple areas or layers that are owned by the different teams working on the internal microservices.

Challenge #3: How to achieve consistency across multiple microservices

- Data owned by each microservice is private to that microservice
- Think of a product price change when you use a product catalog microservice and a basket microservice



Avaelgo

Databases are private per microservice

As stated previously, the data owned by each microservice is private to that microservice and can only be accessed using its microservice API. Therefore, a challenge presented is how to implement end-to-end business processes while keeping consistency across multiple microservices.

To analyze this problem, let's look at an example from the eShopOnContainers reference application. The Catalog microservice maintains information about all the products, including the product price. The Basket microservice manages temporal data about product items that users are adding to their shopping baskets, which includes the price of the items at the time they were added to the basket. When a product's price is updated in the catalog, that price should also be updated in the active baskets that hold that same product, plus the system should probably warn the user saying that a particular item's price has changed since they added it to their basket.

In a hypothetical monolithic version of this application, when the price changes in the products table, the catalog subsystem could simply use an ACID transaction to update the current price in the Basket table.

However, in a microservices-based application, the Product and Basket tables are owned by their respective microservices. No microservice should ever include tables/storage owned by another microservice in its own transactions, not even in direct queries.

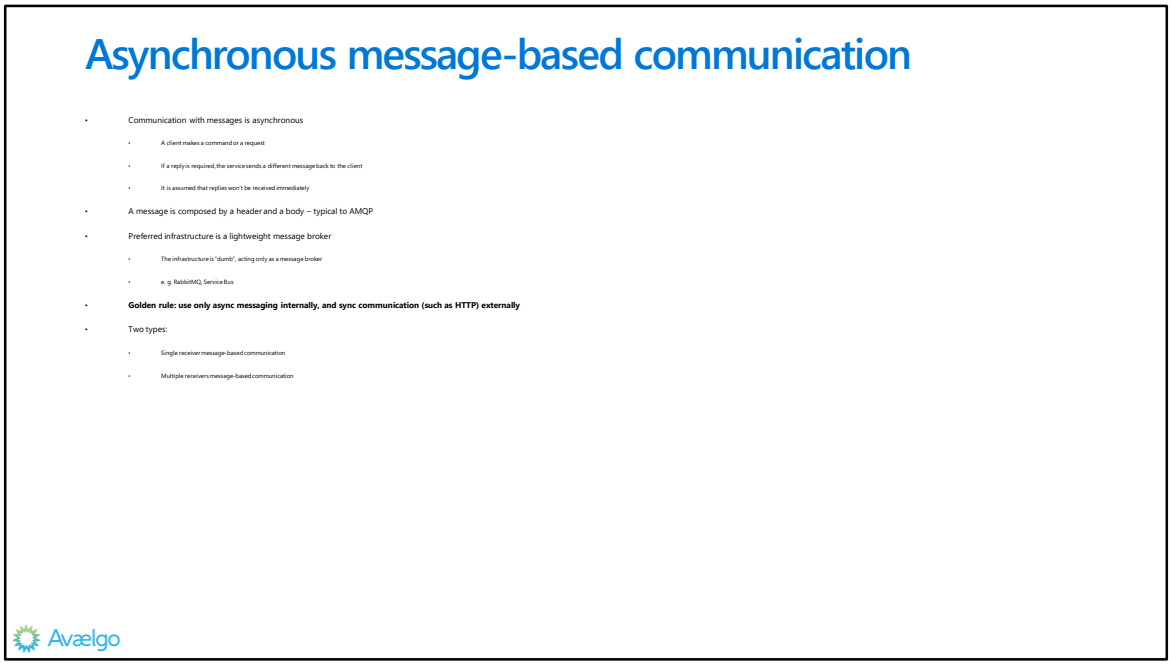
The Catalog microservice shouldn’t update the Basket table directly, because the Basket table is owned by the Basket microservice.

Challenge #3: How to achieve consistency across multiple microservices

- In these cases, always accept **eventual consistency**
- Implement asynchronous communication such as integration events
  - This means that you need to choose between availability and ACID strong consistency
  - Remember the CAP theorem:
    - **Consistency**: Every read receives the most recent write or an error
    - **Availability**: Every request receives a (non-error) response – without the guarantee that it contains the most recent write
    - **Partition tolerance**: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

Avælgo

To make an update to the Basket microservice, the Catalog microservice should use eventual consistency probably based on asynchronous communication such as integration events (message and event-based communication).
As stated by the CAP theorem, you need to choose between availability and ACID strong consistency. Most microservice-based scenarios demand availability and high scalability as opposed to strong consistency. Mission-critical applications must remain up and running, and developers can work around strong consistency by using techniques for working with weak or eventual consistency. This is the approach taken by most microservice-based architectures.
Moreover, ACID-style or two-phase commit transactions are not just against microservices principles; most NoSQL databases (like Azure Cosmos DB, MongoDB, etc.) do not support two-phase commit transactions, typical in distributed databases scenarios. However, maintaining data consistency across services and databases is essential. This challenge is also related to the question of how to propagate changes across multiple microservices when certain data needs to be redundant—for example, when you need to have the product's name or description in the Catalog microservice and the Basket microservice.



Asynchronous messaging and event-driven communication are critical when propagating changes across multiple microservices and their related domain models. As mentioned earlier in the discussion microservices and Bounded Contexts (BCs), models (User, Customer, Product, Account, etc.) can mean different things to different microservices or BCs. That means that when changes occur, you need some way to reconcile changes across the different models. A solution is eventual consistency and event-driven communication based on asynchronous messaging.

When using messaging, processes communicate by exchanging messages asynchronously. A client makes a command or a request to a service by sending it a message. If the service needs to reply, it sends a different message back to the client. Since it's a message-based communication, the client assumes that the reply won't be received immediately, and that there might be no response at all.

A message is composed by a header (metadata such as identification or security information) and a body. Messages are usually sent through asynchronous protocols like AMQP.
The preferred infrastructure for this type of communication in the microservices

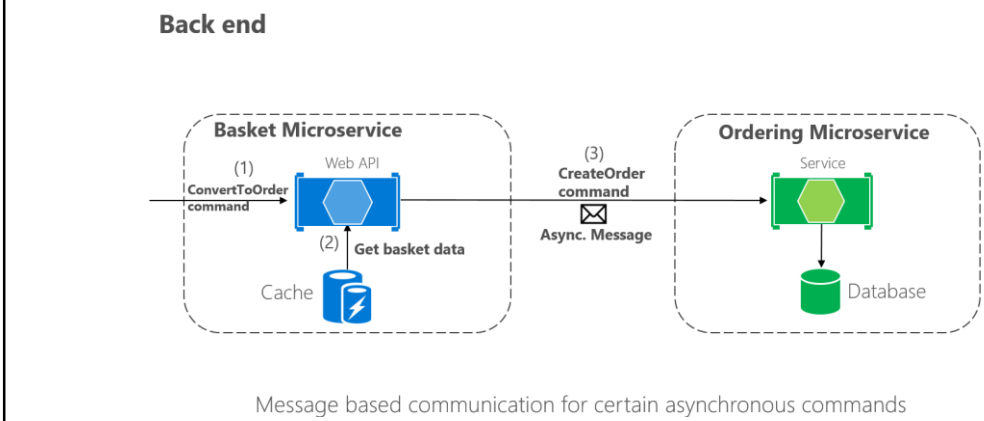
community is a lightweight message broker, which is different than the large brokers and orchestrators used in SOA. In a lightweight message broker, the infrastructure is typically “dumb,” acting only as a message broker, with simple implementations such as RabbitMQ or a scalable service bus in the cloud like Azure Service Bus. In this scenario, most of the “smart” thinking still lives in the endpoints that are producing and consuming messages-that is, in the microservices.

Another rule you should try to follow, as much as possible, is to use only asynchronous messaging between the internal services, and to use synchronous communication (such as HTTP) only from the client apps to the front-end services (API Gateways plus the first level of microservices).

There are two kinds of asynchronous messaging communication: single receiver message-based communication, and multiple receivers message-based communication. The following sections provide details about them.

Asynchronous message-based communication

Single-receiver message-based communication



Message based communication for certain asynchronous commands

Avælgo

Message-based asynchronous communication with a single receiver means there's point-to-point communication that delivers a message to exactly one of the consumers that's reading from the channel, and that the message is processed just once. However, there are special situations. For instance, in a cloud system that tries to automatically recover from failures, the same message could be sent multiple times. Due to network or other failures, the client has to be able to retry sending messages, and the server has to implement an operation to be idempotent in order to process a particular message just once.

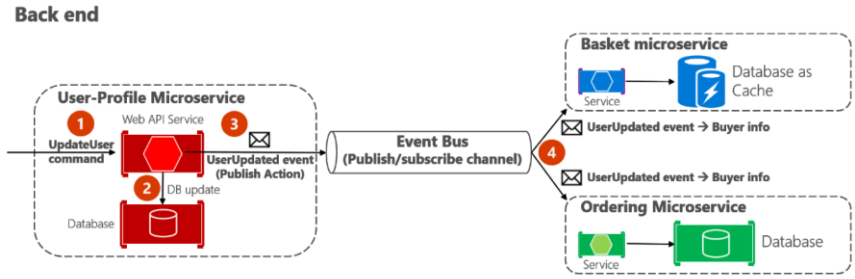
Single-receiver message-based communication is especially well suited for sending asynchronous commands from one microservice to another as shown in Figure 4-18 that illustrates this approach.

Once you start sending message-based communication (either with commands or events), you should avoid mixing message-based communication with synchronous HTTP communication.

Asynchronous message-based communication

Multiple-receivers message-based communication

- Publish/subscribe mechanism
- If eventual consistency is used, this should be made clear to the user too



Eventual consistency across microservices based on event-driven async communication

Avælgo

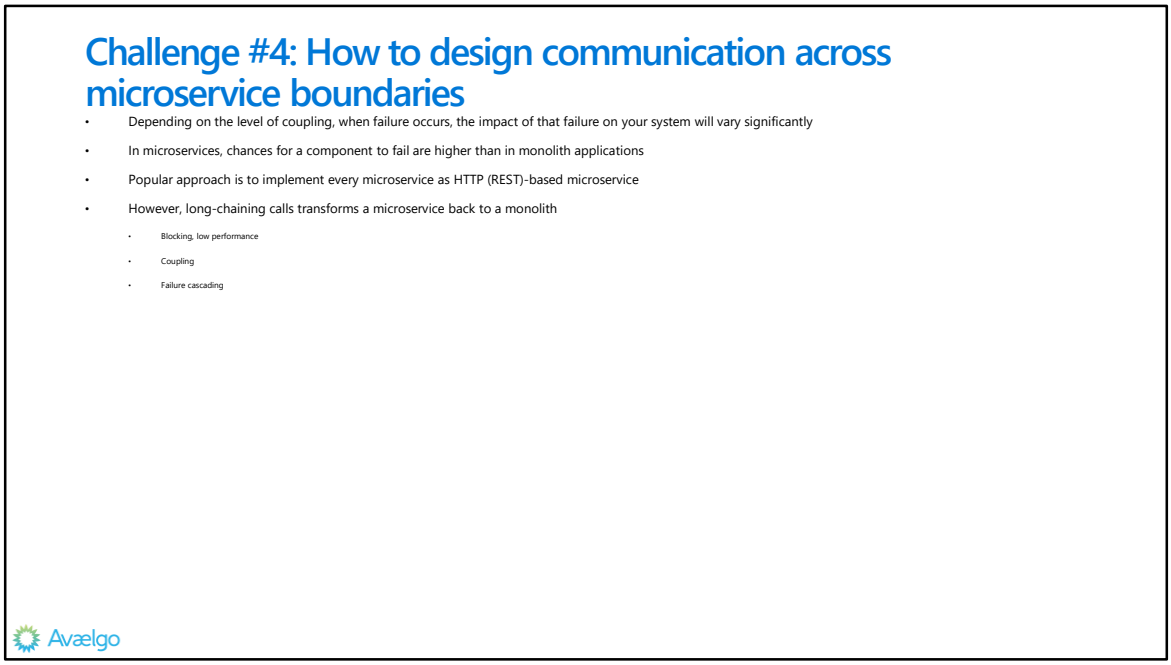
When using asynchronous event-driven communication, a microservice publishes an integration event when something happens within its domain and another microservice needs to be aware of it, like a price change in a product catalog microservice. Additional microservices subscribe to the events so they can receive them asynchronously. When that happens, the receivers might update their own domain entities, which can cause more integration events to be published. This publish/subscribe system is usually performed by using an implementation of an event bus. The event bus can be designed as an abstraction or interface, with the API that's needed to subscribe or unsubscribe to events and to publish events. The event bus can also have one or more implementations based on any inter-process and messaging broker, like a messaging queue or service bus that supports asynchronous communication and a publish/subscribe model.

If a system uses eventual consistency driven by integration events, it's recommended that this approach is made completely clear to the end user. The system shouldn't use an approach that mimics integration events, like SignalR or polling systems from the client. The end user and the business owner have to explicitly embrace eventual consistency in the system and realize that in many cases the business doesn't have any problem with this approach, as long as it's explicit. This is important because users

might expect to see some results immediately and this might not happen with eventual consistency.

As noted earlier in the Challenges and solutions for distributed data management section, you can use integration events to implement business tasks that span multiple microservices. Thus, you'll have eventual consistency between those services. An eventually consistent transaction is made up of a collection of distributed actions. At each action, the related microservice updates a domain entity and publishes another integration event that raises the next action within the same end-to-end business task.

An important point is that you might want to communicate to multiple microservices that are subscribed to the same event. To do so, you can use publish/subscribe messaging based on event-driven communication, as shown in Figure 4-19. This publish/subscribe mechanism isn't exclusive to the microservice architecture. It's similar to the way Bounded Contexts in DDD should communicate, or to the way you propagate updates from the write database to the read database in the Command and Query Responsibility Segregation (CQRS) architecture pattern. The goal is to have eventual consistency between multiple data sources across your distributed system.



Communicating across microservice boundaries is a real challenge. In this context, communication doesn't refer to what protocol you should use (HTTP and REST, AMQP, messaging, and so on). Instead, it addresses what communication style you should use, and especially how coupled your microservices should be. Depending on the level of coupling, when failure occurs, the impact of that failure on your system will vary significantly.

In a distributed system like a microservices-based application, with so many artifacts moving around and with distributed services across many servers or hosts, components will eventually fail. Partial failure and even larger outages will occur, so you need to design your microservices and the communication across them considering the common risks in this type of distributed system.

A popular approach is to implement HTTP (REST)-based microservices, due to their simplicity. An HTTP-based approach is perfectly acceptable; the issue here is related to how you use it. If you use HTTP requests and responses just to interact with your microservices from client applications or from API Gateways, that's fine. But if you create long chains of synchronous HTTP calls across microservices, communicating across their boundaries as if the microservices were objects in a monolithic application, your application will eventually run into problems.

For instance, imagine that your client application makes an HTTP API call to an individual microservice like the Ordering microservice. If the Ordering microservice in turn calls additional microservices using HTTP within the same request/response cycle, you're creating a chain of HTTP calls. It might sound reasonable initially. However, there are important points to consider when going down this path:

- Blocking and low performance. Due to the synchronous nature of HTTP, the original request doesn't get a response until all the internal HTTP calls are finished. Imagine if the number of these calls increases significantly and at the same time one of the intermediate HTTP calls to a microservice is blocked. The result is that performance is impacted, and the overall scalability will be exponentially affected as additional HTTP requests increase.
- Coupling microservices with HTTP. Business microservices shouldn't be coupled with other business microservices. Ideally, they shouldn't "know" about the existence of other microservices. If your application relies on coupling microservices as in the example, achieving autonomy per microservice will be almost impossible
- Failure in any one microservice. If you implemented a chain of microservices linked by HTTP calls, when any of the microservices fails (and eventually they will fail) the whole chain of microservices will fail. A microservice-based system should be designed to continue to work as well as possible during partial failures. Even if you implement client logic that uses retries with exponential backoff or circuit breaker mechanisms, the more complex the HTTP call chains are, the more complex it is to implement a failure strategy based on HTTP.

Challenge #4: How to design communication across microservice boundaries

- Alternative is to use asynchronous interaction for inter-microservice communication
- Either use asynchronous message and event-based communication
- Or use HTTP polling independently of the original HTTP request/response cycle

Therefore, in order to enforce microservice autonomy and have better resiliency, you should minimize the use of chains of request/response communication across microservices. It's recommended that you use only asynchronous interaction for inter-microservice communication, either by using asynchronous message- and event-based communication, or by using (asynchronous) HTTP polling independently of the original HTTP request/response cycle.

References

- CAP theorem
  - https://en.wikipedia.org/wiki/CAP_theorem
- Eventual consistency
  - https://en.wikipedia.org/wiki/Eventual_consistency
- Data Consistency Primer
  - https://docs.microsoft.com/previous-versions/msp-n-p/dn589800(v=pandp.10)
- Martin Fowler. CQRS (Command and Query Responsibility Segregation)
  - https://martinfowler.com/bliki/CQRS.html

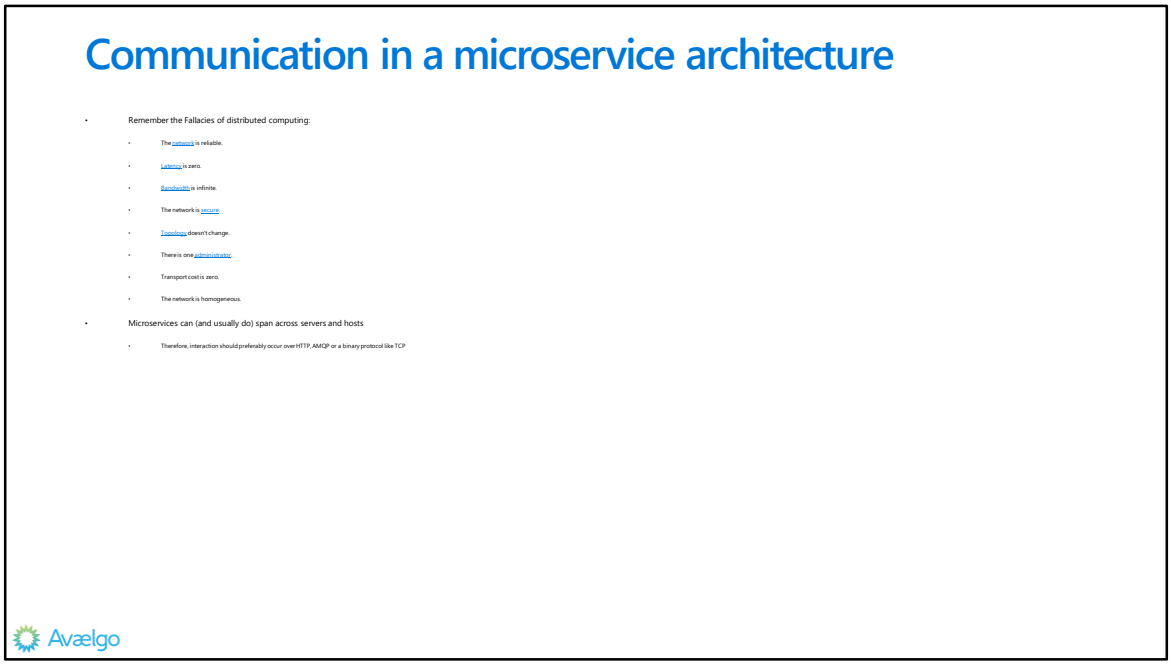
References

- Materialized View
  - https://docs.microsoft.com/azure/architecture/patterns/materialized-view
- Charles Row. ACID vs. BASE
  - http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/
- Compensating Transactions
  - https://docs.microsoft.com/azure/architecture/patterns/compensating-transaction
- Udi Dahan. Service Oriented Composition
  - http://udidahan.com/2014/07/30/service-oriented-composition-with-video/

Avælgo

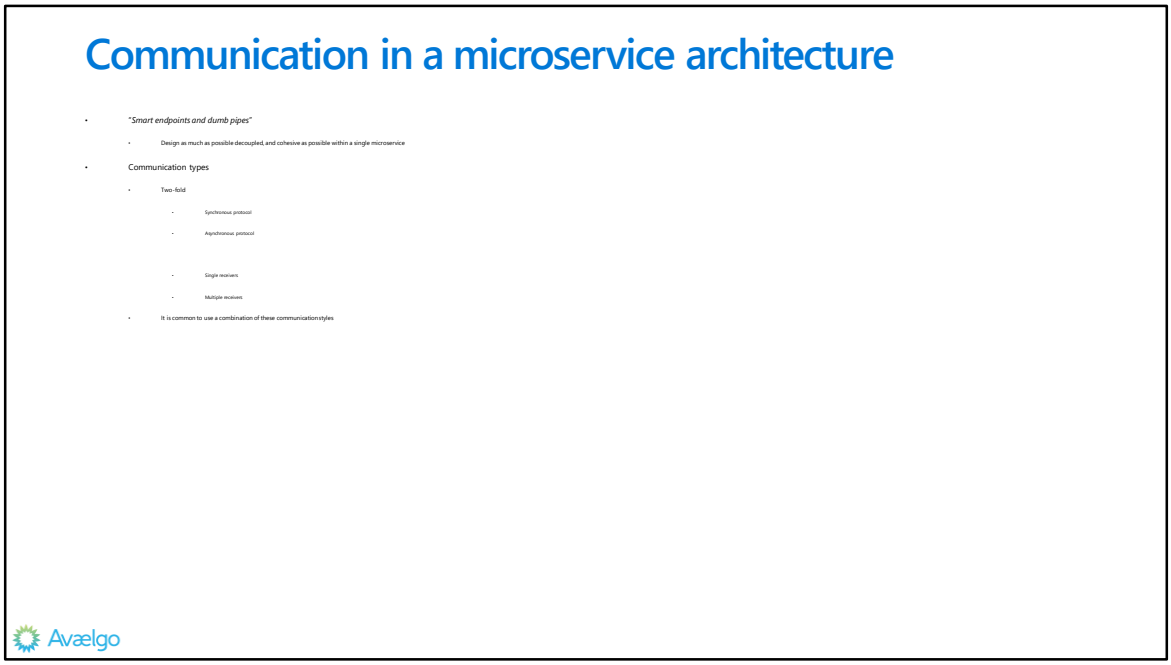
Demystifying Communication in a microservice architecture

TODO P 49 → P 54



In a monolithic application running on a single process, components invoke one another using language-level method or function calls. These can be strongly coupled if you're creating objects with code (for example, new ClassName()), or can be invoked in a decoupled way if you're using Dependency Injection by referencing abstractions rather than concrete object instances. Either way, the objects are running within the same process. The biggest challenge when changing from a monolithic application to a microservices-based application lies in changing the communication mechanism. A direct conversion from in-process method calls into RPC calls to services will cause a chatty and not efficient communication that won't perform well in distributed environments. The challenges of designing distributed system properly are well enough known that there's even a canon known as the Fallacies of distributed computing that lists assumptions that developers often make when moving from monolithic to distributed designs

A microservices-based application is a distributed system running on multiple processes or services, usually even across multiple servers or hosts. Each service instance is typically a process. Therefore, services must interact using an inter-process communication protocol such as HTTP, AMQP, or a binary protocol like TCP, depending on the nature of each service.



The microservice community promotes the philosophy of “smart endpoints and dumb pipes” This slogan encourages a design that’s as decoupled as possible between microservices, and as cohesive as possible within a single microservice. As explained earlier, each microservice owns its own data and its own domain logic. But the microservices composing an end-to-end application are usually simply choreographed by using REST communications rather than complex protocols such as WS-* and flexible event-driven communications instead of centralized business-process-orchestrators.

Client and services can communicate through many different types of communication, each one targeting a different scenario and goals. Initially, those types of communications can be classified in two axes.
The first axis defines if the protocol is synchronous or asynchronous:
• Synchronous protocol. HTTP is a synchronous protocol. The client sends a request and waits for a response from the service. That’s independent of the client code execution that could be synchronous (thread is blocked) or asynchronous (thread isn’t blocked, and the response will reach a callback eventually). The important point here is that the protocol (HTTP/HTTPS) is synchronous and the client code can only continue its task when it receives the HTTP server response.

• Asynchronous protocol. Other protocols like AMQP (a protocol supported by many operating systems and cloud environments) use asynchronous messages. The client code or message sender usually doesn't wait for a response. It just sends the message as when sending a message to a RabbitMQ queue or any other message broker.

The second axis defines if the communication has a single receiver or multiple receivers:
• Single receiver. Each request must be processed by exactly one receiver or service. An example of this communication is the Command pattern.
• Multiple receivers. Each request can be processed by zero to multiple receivers. This type of communication must be asynchronous. An example is the publish/subscribe mechanism used in patterns like Event-driven architecture. This is based on an event-bus interface or message broker when propagating data updates between multiple microservices through events; it's usually implemented through a service bus or similar artifact like Azure Service Bus by using topics and subscriptions.

Async microservice integration enforces autonomy

- The fewer communications between microservices, the better
- Communication between microservices should be done only by propagating data asynchronously
- Never depend on synchronous communication between microservices, **not even for queries**
  - Otherwise, you may have an architecture which is not resilient to fails
- **The more synchronous dependencies an app has, the worse the overall response time gets to the client app**

Avælgo

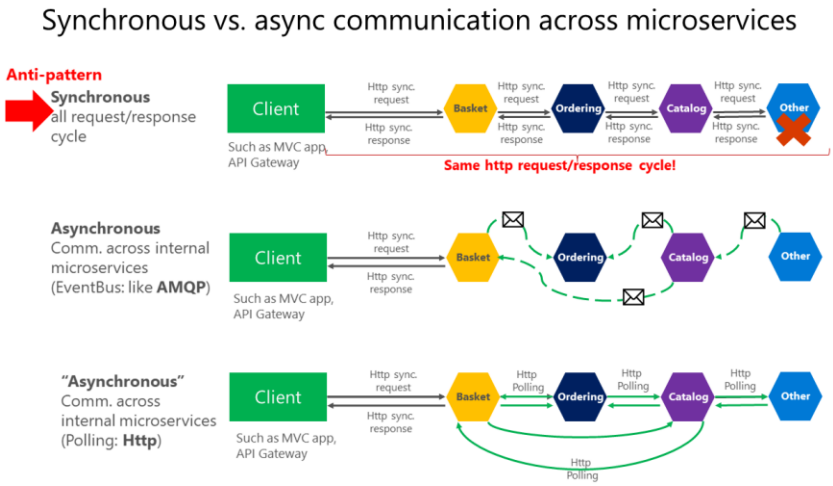
As mentioned, the important point when building a microservices-based application is the way you integrate your microservices. Ideally, you should try to minimize the communication between the internal microservices. The fewer communications between microservices, the better. But in many cases, you'll have to somehow integrate the microservices. When you need to do that, the critical rule here is that the communication between the microservices should be asynchronous. That doesn't mean that you have to use a specific protocol (for example, asynchronous messaging versus synchronous HTTP). It just means that the communication between microservices should be done only by propagating data asynchronously, but try not to depend on other internal microservices as part of the initial service's HTTP request/response operation.

If possible, never depend on synchronous communication (request/response) between multiple microservices, not even for queries. The goal of each microservice is to be autonomous and available to the client consumer, even if the other services that are part of the end-to-end application are down or unhealthy. If you think you need to make a call from one microservice to other microservices (like performing an HTTP request for a data query) to be able to provide a response to a client application, you have an architecture that won't be resilient when some microservices fail.

Moreover, having HTTP dependencies between microservices, like when creating long request/response cycles with HTTP request chains, as shown in the first part of the Figure 4-15, not only makes your microservices not autonomous but also their performance is impacted as soon as one of the services in that chain isn't performing well.

The more you add synchronous dependencies between microservices, such as query requests, the worse the overall response time gets for the client apps.

Synchronous vs. async communication across microservices

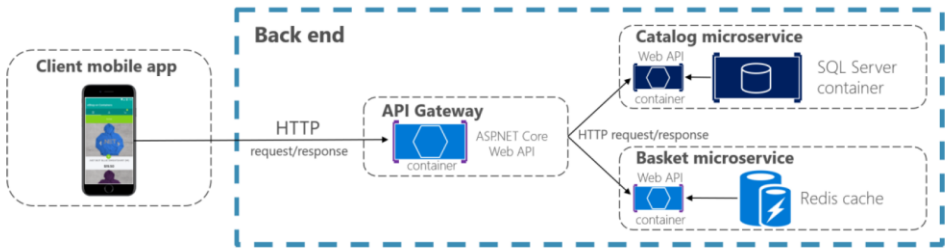


Avælgo

If your microservice needs to raise an additional action in another microservice, if possible, do not perform that action synchronously and as part of the original microservice request and reply operation. Instead, do it asynchronously (using asynchronous messaging or integration events, queues, etc.). But, as much as possible, do not invoke the action synchronously as part of the original synchronous request and reply operation.

And finally (and this is where most of the issues arise when building microservices), if your initial microservice needs data that's originally owned by other microservices, do not rely on making synchronous requests for that data. Instead, replicate or propagate that data (only the attributes you need) into the initial service's database by using eventual consistency

Request/response communication with HTTP and REST



Avælgo

When a client uses request/response communication, it sends a request to a service, then the service processes the request and sends back a response. Request/response communication is especially well suited for querying data for a real-time UI (a live user interface) from client apps. Therefore, in a microservice architecture you'll probably use this communication mechanism for most queries.

When a client uses request/response communication, it assumes that the response will arrive in a short time, typically less than a second, or a few seconds at most. For delayed responses, you need to implement asynchronous communication based on messaging patterns and messaging technologies.

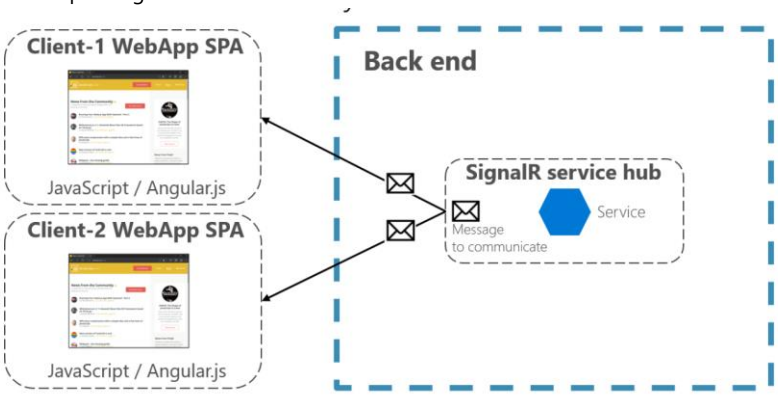
A popular architectural style for request/response communication is REST. This approach is based on, and tightly coupled to, the HTTP protocol, embracing HTTP verbs like GET, POST, and PUT. REST is the most commonly used architectural communication approach when creating services. You can implement REST services when you develop ASP.NET Core Web API services.

There's additional value when using HTTP REST services as your interface definition language. For instance, if you use Swagger metadata to describe your service API, you

can use tools that generate client stubs that can directly discover and consume your services.

Push and real-time communication based on HTTP

- Real-time HTTP communication means that
  - Server code pushing content to connected clients



Avaelgo

Another possibility (usually for different purposes than REST) is a real-time and one-to-many communication with higher-level frameworks such as ASP.NET SignalR and protocols such as WebSockets.
Since communication is in real time, client apps show the changes almost instantly. This is usually handled by a protocol such as WebSockets, using many WebSockets connections (one per client). A typical example is when a service communicates a change in the score of a sports game to many client web apps simultaneously.

Resources

- Event Driven Messaging
  - http://soapatterns.org/design_patterns/event_driven_messaging
- Publish/Subscribe Channel
  - https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html
- Udi Dahan. Clarified CQRS
  - http://udidahan.com/2009/12/09/clarified-cqrs/
- Command and Query Responsibility Segregation (CQRS)
  - https://docs.microsoft.com/azure/architecture/patterns/cqrs
- Communicating Between Bounded Contexts
  - https://docs.microsoft.com/previous-versions/msp-n-p/jj591572(v=pandp.10)
- Eventual consistency
  - https://en.wikipedia.org/wiki/Eventual_consistency
- Jimmy Bogard. Refactoring Towards Resilience: Evaluating Coupling
  - https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/

Avælgo

Microservice addressability using a Service Registry

TODO P 59 → P 61

Microservices addressability and the service registry

- Each microservice has a unique name (URL)
- If you have to think about which computer is running a particular microservices, things will go bad quickly
- Microservices need addressable names that make them independent from the infrastructure that they're running on
  - Implies interaction btw. how a service is deployed and how it's discovered
- The registry is a database containing the network locations of service instances

Each microservice has a unique name (URL) that's used to resolve its location. Your microservice needs to be addressable wherever it's running. If you have to think about which computer is running a particular microservice, things can go bad quickly. In the same way that DNS resolves a URL to a particular computer, your microservice needs to have a unique name so that its current location is discoverable. Microservices need addressable names that make them independent from the infrastructure that they're running on. This implies that there's an interaction between how your service is deployed and how it's discovered, because there needs to be a service registry. In the same vein, when a computer fails, the registry service must be able to indicate where the service is now running.

The service registry pattern is a key part of service discovery. The registry is a database containing the network locations of service instances. A service registry needs to be highly available and up-to-date. Clients could cache network locations obtained from the service registry. However, that information eventually goes out of date and clients can no longer discover service instances. Consequently, a service registry consists of a cluster of servers that use a replication protocol to maintain consistency

Resources

- Chris Richardson. Pattern: Service registry
  - https://microservices.io/patterns/service-registry.html
- Auth0. The Service Registry
  - https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry
- Gabriel Schenker. Service discovery
  - https://lostechies.com/gabrielschenker/2016/01/27/service-discovery/

Composite UI microservices for real-world web applications

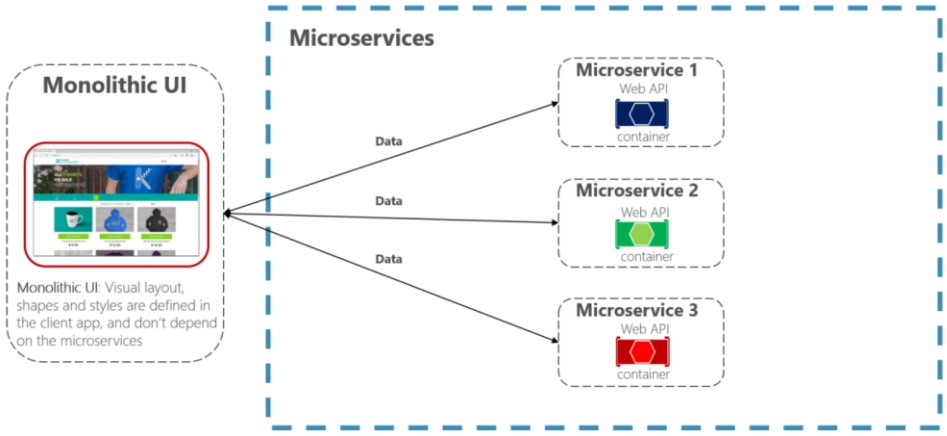
TODO P 61 → P 62

Creating composite UI based on microservices

- Microservices tend to start with the server-side handling data and logic
  - A more advanced approach is to design your application UI based on microservices as well
- A composite UI produced by the microservices, instead of having microservices on the server and just a monolithic client app

Microservices architecture often starts with the server-side handling data and logic. However, a more advanced approach is to design your application UI based on microservices as well. That means having a composite UI produced by the microservices, instead of having microservices on the server and just a monolithic client app consuming the microservices. With this approach, the microservices you build can be complete with both logic and visual representation

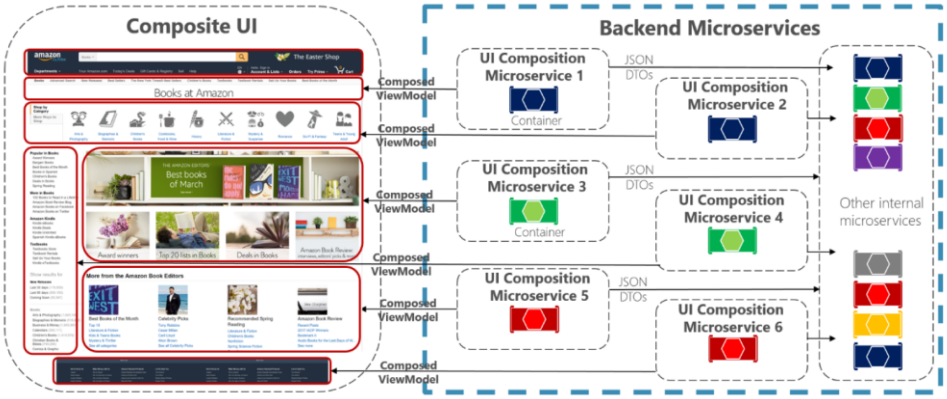
Creating composite UI based on microservices



Avælgo

The figure shows the simpler approach of just consuming microservices from a monolithic client application. Of course, you could have an ASP.NET MVC service in between producing the HTML and JavaScript. The figure is a simplification that highlights that you have a single (monolithic) client UI consuming the microservices, which just focus on logic and data and not on the UI shape (HTML and JavaScript).

Creating composite UI based on microservices



Avælgo

In contrast, a composite UI is precisely generated and composed by the microservices themselves. Some of the microservices drive the visual shape of specific areas of the UI. The key difference is that you have client UI components (TypeScript classes, for example) based on templates, and the data-shaping-UI ViewModel for those templates comes from each microservice

At client application start-up time, each of the client UI components (TypeScript classes, for example) registers itself with an infrastructure microservice capable of providing ViewModels for a given scenario. If the microservice changes the shape, the UI changes also.

A composite UI approach that's driven by microservices can be more challenging or less so, depending on what UI technologies you're using. For instance, you won't use the same techniques for building a traditional web application that you use for building an SPA or for native mobile app (as when developing Xamarin apps, which can be more challenging for this approach).

Resources

- Composite UI using ASP.NET (Particular's Workshop)
  - https://github.com/Particular/Workshop/tree/master/demos/asp-net-core
- Ruben Oostinga. The Monolithic Frontend in the Microservices Architecture
  - https://blog.xebia.com/the-monolithic-frontend-in-the-microservices-architecture/
- Mauro Servienti. The secret of better UI composition
  - https://particular.net/blog/secret-of-better-ui-composition
- Viktor Farcic. Including Front-End Web Components Into Microservices
  - https://technologyconversations.com/2015/08/09/including-front-end-web-components-into-microservices/
- Managing Frontend in the Microservices Architecture
  - https://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html

Avælgo

Applying simplified CQRS and DDD patterns in a microservice

TODO P 184 → P 236

What is CQRS?

- CQRS is related to CQS
  - Basic idea is to divide a system's operations into two sharply separated categories
    - Queries
    - Commands
- CQRS
  - Unlike CQS, it is a pattern based on commands, events and asynchronous messages
  - Different database for reads than for writes
  - Implements event-sourcing

Avælgo

is an architectural pattern that separates the models for reading and writing data. The related term Command Query Separation (CQS) was originally defined by Bertrand Meyer in his book *Object Oriented Software Construction*. The basic idea is that you can divide a system's operations into two sharply separated categories:
• Queries. These return a result and do not change the state of the system, and they are free of side effects.
• Commands. These change the state of a system.

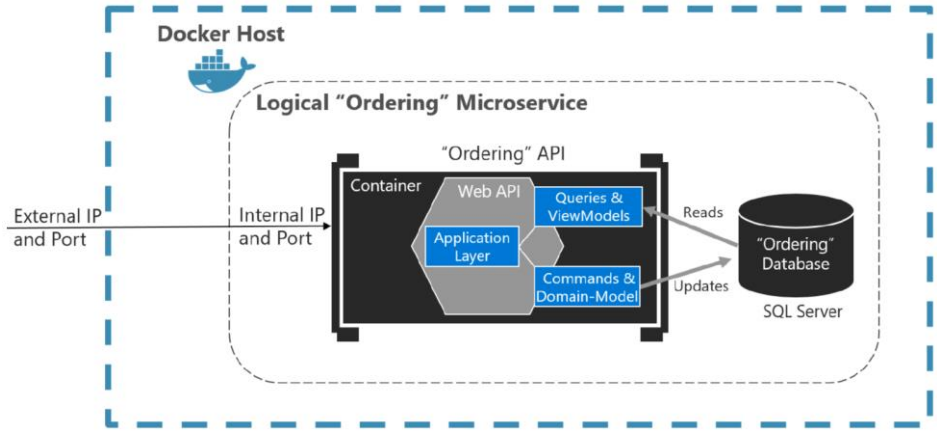
CQS is a simple concept—it is about methods within the same object being either queries or commands. Each method either returns state or mutates state, but not both. Even a single repository pattern object can comply with CQS. CQS can be considered a foundational principle for CQRS.

Command and Query Responsibility Segregation (CQRS) was introduced by Greg Young and strongly promoted by Udi Dahan and others. It is based on the CQS principle, although it is more detailed. It can be considered a pattern based on commands and events plus optionally on asynchronous messages. In many cases, CQRS is related to more advanced scenarios, like having a different physical database for reads (queries) than for writes (updates). Moreover, a more evolved CQRS system might implement

Event-Sourcing (ES) for your updates database, so you would only store events in the domain model instead of storing the current-state data. However, this is not the approach used in this guide; we are using the simplest CQRS approach, which consists of just separating the queries from the commands.

The separation aspect of CQRS is achieved by grouping query operations in one layer and commands in another layer. Each layer has its own data model (note that we say model, not necessarily a different database) and is built using its own combination of patterns and technologies. More importantly, the two layers can be within the same tier or microservice, as in the example (ordering microservice) used for this guide. Or they could be implemented on different microservices or processes so they can be optimized and scaled out separately without affecting one another.

Simplified CQRS and DDD microservice



Avælgo

CQRS means having two objects for a read/write operation where in other contexts there is one. There are reasons to have a denormalized reads database, which you can learn about in more advanced CQRS literature. But we are not using that approach here, where the goal is to have more flexibility in the queries instead of limiting the queries with constraints from DDD patterns like aggregates.
An example of this kind of service is the ordering microservice from the eShopOnContainers reference application. This service implements a microservice based on a simplified CQRS approach. It uses a single data source or database, but two logical models plus DDD patterns for the transactional domain,
The application layer can be the Web API itself. The important design aspect here is that the microservice has split the queries and ViewModels (data models especially created for the client applications) from the commands, domain model, and transactions following the CQRS pattern. This approach keeps the queries independent from restrictions and constraints coming from DDD patterns that only make sense for transactions and updates, as explained in later sections.

Notes

- The essence of those patterns, and the important point here, is that queries are idempotent: no matter how many times you query a system, the state of that system will not change
- On the other hand, commands, which trigger transactions and data updates, change state in the system

Avælgo

The essence of those patterns, and the important point here, is that queries are idempotent: no matter how many times you query a system, the state of that system will not change.

On the other hand, commands, which trigger transactions and data updates, change state in the system. With commands, you need to be careful when dealing with complexity and ever-changing business rules. This is the where you want to apply DDD techniques to have a better modeled system.

Use ViewModels specifically made for client apps

- Data Transfer Objects (DTOs) are called ViewModels, when queries are performed to obtain data needed by client apps and returned to them
- A ViewModel can be the result of joining data from multiple entities
- ViewModels should be static
  - Dynamic objects can still be called ViewModels, but:
    - Whilst allowing for faster development, compatibility will have to suffer and Swashbubble cannot provide the same level of documentation

Avælgo

Tips

- Always describe the response type of Web APIs

```
[Route("api/v1/[controller]")]
[Authorize]
public class OrdersController : Controller
{
    //Additional code...
    [Route("")]
    [HttpGet]
    [ProducesResponseType(typeof(IEnumerable<OrderSummary>),
        (int)HttpStatusCode.OK)]
    public async Task<IActionResult> GetOrders()
    {
        var userid = _identityService.GetUserIdentity();
        var orders = await _orderQueries.GetOrdersFromUserAsync(Guid.Parse(userid));
        return Ok(orders);
    }
}
}
```

Avælgo

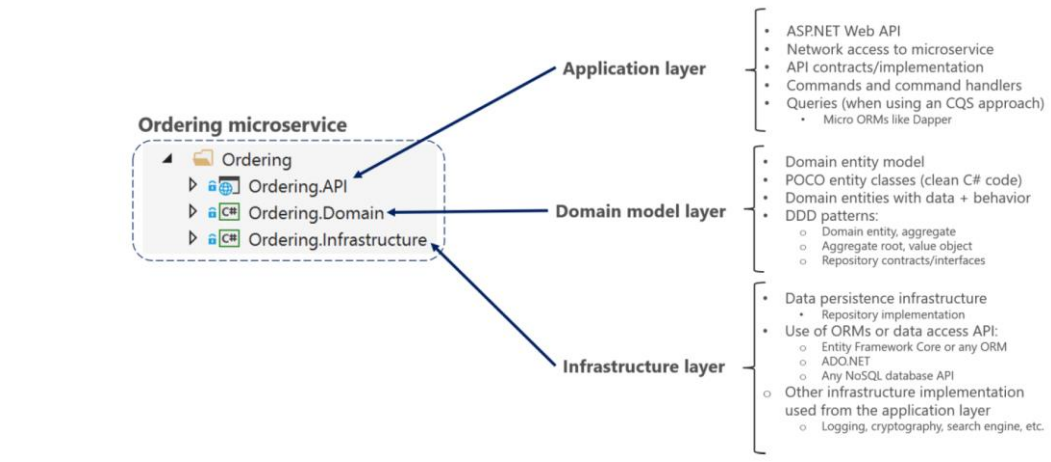
Swagger integration

```
public class OrderSummary
{
    public int ordernumber { get; set; }
    public DateTime date { get; set; }
    public string status { get; set; }
    public double total { get; set; }
}
```

Avælgo

However, the ProducesResponseType attribute cannot use dynamic as a type but requires to use explicit types, like the OrderSummary ViewModel DTO, shown in the following example.
This is another reason why explicit returned types are better than dynamic types, in the long term. When using the ProducesResponseType attribute, you can also specify what is the expected outcome in regards possible HTTP errors/codes, like 200, 400, etc.

Layers in a Domain-Driven Design Microservice



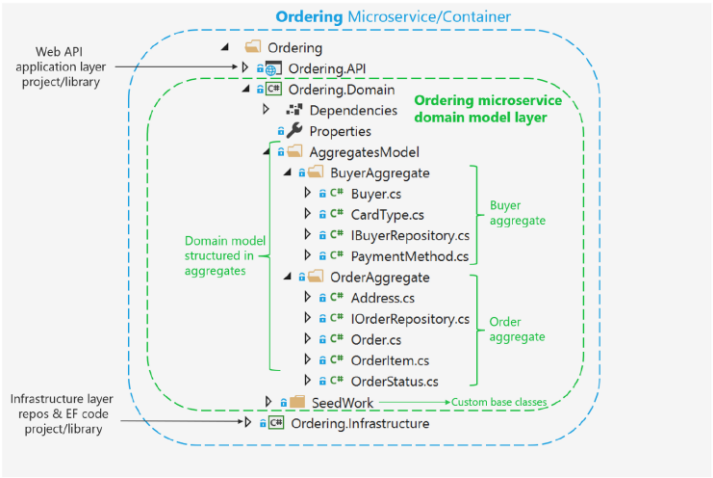
Avælgo

Most enterprise applications with significant business and technical complexity are defined by multiple layers. The layers are a logical artifact, and are not related to the deployment of the service. They exist to help developers manage the complexity in the code. Different layers (like the domain model layer versus the presentation layer, etc.) might have different types, which mandates translations between those types.

For example, an entity could be loaded from the database. Then part of that information, or an aggregation of information including additional data from other entities, can be sent to the client UI through a REST Web API. The point here is that the domain entity is contained within the domain model layer and should not be propagated to other areas that it does not belong to, like to the presentation layer.

When tackling complexity, it is important to have a domain model controlled by aggregate roots that make sure that all the invariants and rules related to that group of entities (aggregate) are performed through a single entry-point or gate, the aggregate root.

Domain model structure in a custom .NET Standard Library

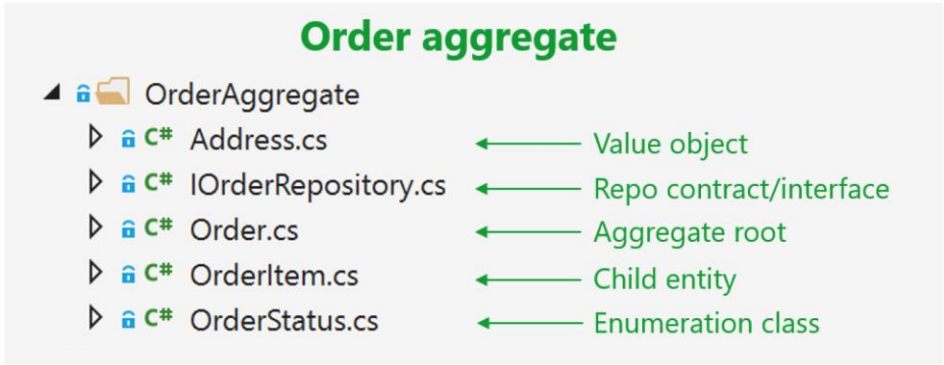


Avælgo

The folder organization used for the eShopOnContainers reference application demonstrates the DDD model for the application. You might find that a different folder organization more clearly communicates the design choices made for your application. As you can see in Figure 7-10, in the ordering domain model there are two aggregates, the order aggregate and the buyer aggregate. Each aggregate is a group of domain entities and value objects, although you could have an aggregate composed of a single domain entity (the aggregate root or root entity) as well.

Additionally, the domain model layer includes the repository contracts (interfaces) that are the infrastructure requirements of your domain model. In other words, these interfaces express what repositories and the methods the infrastructure layer must implement. It is critical that the implementation of the repositories be placed outside of the domain model layer, in the infrastructure layer library, so the domain model layer is not “contaminated” by API or classes from infrastructure technologies, like Entity Framework.

Structure aggregates in a custom .NET Standard library



Avælgo

An aggregate refers to a cluster of domain objects grouped together to match transactional consistency. Those objects could be instances of entities (one of which is the aggregate root or root entity) plus any additional value objects.
Transactional consistency means that an aggregate is guaranteed to be consistent and up to date at the end of a business action.

Handling failure and partial failure and implementing retries with exponential backoff

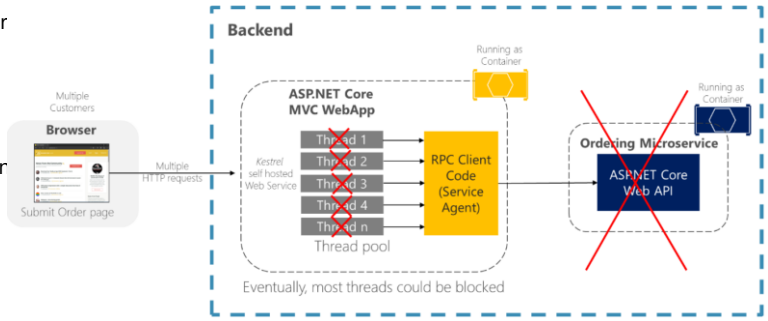
Avælgo

TODO P 284 → P 287

Partial failure is an ever-present risk

A single microservice can fail or might not be available to respond

Since clients and services are separate processes, a service might not be able to respond in a timely way

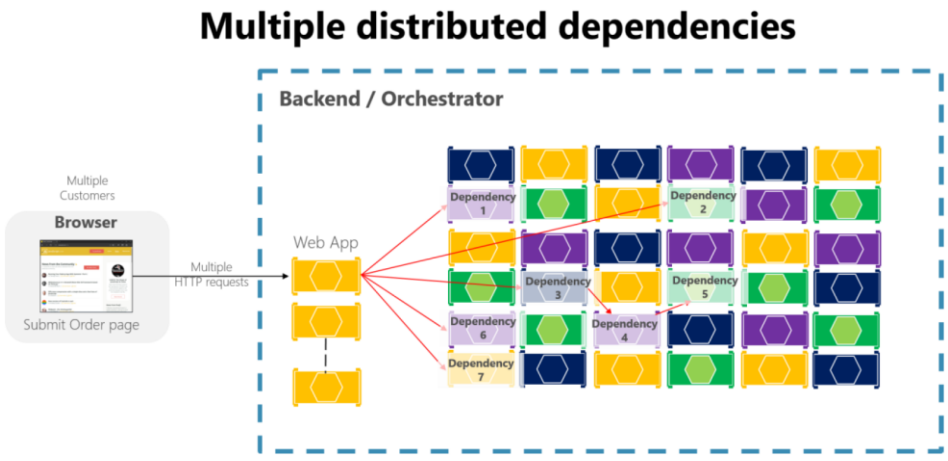


Avaelgo

In distributed systems like microservices-based applications, there's an ever-present risk of partial failure. For instance, a single microservice/container can fail or might not be available to respond for a short time, or a single VM or server can crash. Since clients and services are separate processes, a service might not be able to respond in a timely way to a client's request. The service might be overloaded and responding very slowly to requests or might simply not be accessible for a short time because of network issues.

For example, if the ordering microservice is unresponsive when the user tries to submit an order, a bad implementation of the client process (the MVC web application)—for example, if the client code were to use synchronous RPCs with no timeout—would block threads indefinitely waiting for a response. Besides creating a bad user experience, every unresponsive wait consumes or blocks a thread, and threads are extremely valuable in highly scalable applications. If there are many blocked threads, eventually the application's runtime can run out of threads. In that case, the application can become globally unresponsive instead of just partially unresponsive.

Failures when HTTP chaining is used

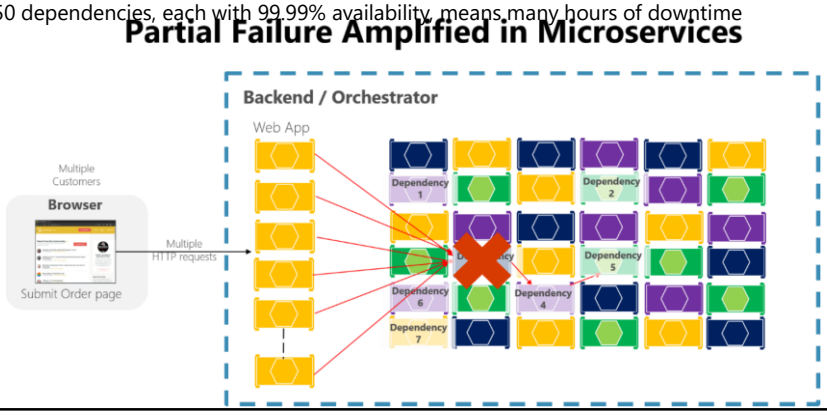


Avaelgo

In a large microservices-based application, any partial failure can be amplified, especially if most of the internal microservices interaction is based on synchronous HTTP calls (which is considered an anti-pattern). Think about a system that receives millions of incoming calls per day. If your system has a bad design that's based on long chains of synchronous HTTP calls, these incoming calls might result in many more millions of outgoing calls (let's suppose a ratio of 1:4) to dozens of internal microservices as synchronous dependencies.

Considerations

- Intermittent failure is guaranteed in a distributed system
- Design to ensure fault-tolerance
  - Small downtimes can be amplified otherwise
  - e. g. 50 dependencies, each with 99.99% availability, means many hours of downtime



Avælgo

Handling strategies

- Use asynchronous communication (for example, message-based communication) across internal microservices
- Use retries with exponential backoff
- Never block indefinitely
- Use the Circuit Breaker pattern
- Provide fallbacks
- Limit the number of queued requests

Avælgo

Strategies for dealing with partial failures include the following.

Use asynchronous communication (for example, message-based communication) across internal microservices. It's highly advisable not to create long chains of synchronous HTTP calls across the internal microservices because that incorrect design will eventually become the main cause of bad outages. On the contrary, except for the front-end communications between the client applications and the first level of microservices or fine-grained API Gateways, it's recommended to use only asynchronous (message-based) communication once past the initial request/response cycle, across the internal microservices. Eventual consistency and event-driven architectures will help to minimize ripple effects. These approaches enforce a higher level of microservice autonomy and therefore prevent against the problem noted here.

Use retries with exponential backoff. This technique helps to avoid short and intermittent failures by performing call retries a certain number of times, in case the service was not available only for a short time. This might occur due to intermittent network issues or when a microservice/container is moved to a different node in a cluster. However, if these retries are not designed properly with circuit breakers, it can aggravate the ripple effects, ultimately even causing a Denial of Service (DoS).

Work around network timeouts. In general, clients should be designed not to block indefinitely and to always use timeouts when waiting for a response. Using timeouts ensures that resources are never tied up indefinitely.

Use the Circuit Breaker pattern. In this approach, the client process tracks the number of failed requests. If the error rate exceeds a configured limit, a "circuit breaker" trips so that further attempts fail immediately. (If a large number of requests are failing, that suggests the service is unavailable and that sending requests is pointless.) After a timeout period, the client should try again and, if the new requests are successful, close the circuit breaker.

Provide fallbacks. In this approach, the client process performs fallback logic when a request fails, such as returning cached data or a default value. This is an approach suitable for queries, and is more complex for updates or commands.

Limit the number of queued requests. Clients should also impose an upper bound on the number of outstanding requests that a client microservice can send to a particular service. If the limit has been reached, it's probably pointless to make additional requests, and those attempts should fail immediately. In terms of implementation, the Polly Bulkhead Isolation policy can be used to fulfill this requirement. This approach is essentially a parallelization throttle with SemaphoreSlim as the implementation. It also permits a "queue" outside the bulkhead. You can proactively shed excess load even before execution (for example, because capacity is deemed full). This makes its response to certain failure scenarios faster than a circuit breaker would be, since the circuit breaker waits for the failures. The BulkheadPolicy object in Polly exposes how full the bulkhead and queue are, and offers events on overflow so can also be used to drive automated horizontal scaling.

Resources

- Resiliency patterns
  - https://docs.microsoft.com/azure/architecture/patterns/category/resiliency
- Adding Resilience and Optimizing Performance
  - https://msdn.microsoft.com/library/jj591574.aspx
- Bulkhead. GitHub repo. Implementation with Polly policy.
  - https://github.com/App-vNext/Polly/wiki/Bulkhead
- Designing resilient applications for Azure
  - https://docs.microsoft.com/azure/architecture/resiliency/
- Transient fault handling
  - https://docs.microsoft.com/azure/architecture/best-practices/transient-faults

Avælgo

Implementing resilient HTTP calls using HttpClientFactory and Polly

Avælgo

Implement HTTP call retries with exponential backoff with HttpClientFactory and Polly policies

- It is recommended to take advantage of more advanced .NET libraries like Polly
- Polly is a .NET library that provides resilience and transient-fault handling capabilities
  - You can implement those capabilities by applying Polly policies:
    - Retry
    - Circuit Breaker
    - Isolation
    - Timeout
    - Fallback

Avælgo

The recommended approach for retries with exponential backoff is to take advantage of more advanced .NET libraries like the open-source Polly library.
Polly is a .NET library that provides resilience and transient-fault handling capabilities. You can implement those capabilities by applying Polly policies such as Retry, Circuit Breaker, Bulkhead Isolation, Timeout, and Fallback. Polly targets .NET 4.x and the .NET Standard Library 1.0 (which supports .NET Core).

Implementation steps

- Using Polly's library with your own custom code with HttpClient can be significantly complex
- Steps:
  1. Reference the ASP.NET Core 2.2 packages
  2. Configure a client with Polly's Retry policy, in startup

```
//ConfigureServices()  - Startup.cs
services.AddHttpClient<IBasketService, BasketService>()
        .SetHandlerLifetime(TimeSpan.FromMinutes(5))  //Set lifetime to five minutes
        .AddPolicyHandler(GetRetryPolicy());
```

```
static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.NotFound)
        .WaitAndRetryAsync(6, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,
                                                                retryAttempt)));
}
```

Avælgo

Reference the ASP.NET Core 2.2 packages
HttpClientFactory is available since .NET Core 2.1 however we recommend you to use the latest ASP.NET Core 2.2 packages from NuGet in your project. You typically need the AspNetCore metapackage, and the extension package Microsoft.Extensions.Http.Polly.

Configure a client with Polly's Retry policy, in Startup
Yneed to define a named or typed client HttpClient configuration in your standard Startup.ConfigureServices(…) method, but now, you add incremental code specifying the policy for the Http retries with exponential backoff.
The **AddPolicyHandler()** method is what adds policies to the HttpClient objects you'll use. In this case, it's adding a Polly's policy for Http Retries with exponential backoff.
To have a more modular approach, the Http Retry Policy can be defined in a separate method within the Startup.cs file.
With Polly, you can define a Retry policy with the number of retries, the exponential backoff configuration, and the actions to take when there's an HTTP exception, such as logging the error. In this case, the policy is configured to try six times with an exponential retry, starting at two seconds.
so it will try six times and the seconds between each retry will be exponential, starting

on two seconds.

Add a jitter strategy to the retry policy

```
Random jitterer = new Random();
Policy
  .Handle<HttpResponseException>() // etc
  .WaitAndRetry(5,    // exponential back-off plus some jitter
      retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
                    + TimeSpan.FromMilliseconds(jitterer.Next(0, 100))
  );
```

Avælgo

A regular Retry policy can impact your system in cases of high concurrency and scalability and under high contention. To overcome peaks of similar retries coming from many clients in case of partial outages, a good workaround is to add a jitter strategy to the retry algorithm/policy. This can improve the overall performance of the end-to-end system by adding randomness to the exponential backoff. This spreads out the spikes when issues arise. When you use a plain Polly policy, code to implement jitter could look like the following example

Resources

- Retry pattern
  - https://docs.microsoft.com/azure/architecture/patterns/retry
- Polly and HttpClientFactory
  - https://github.com/App-vNext/Polly/wiki/Polly-and-HttpClientFactory
- Polly (.NET resilience and transient-fault-handling library)
  - https://github.com/App-vNext/Polly
- Marc Brooker. Jitter: Making Things Better With Randomness
  - https://brooker.co.za/blog/2015/03/21/backoff.html

Avælgo

Implementing the Circuit Breaker pattern

Implement the Circuit Breaker pattern

- When failure occurs, hammering with retries is the last thing you want to do
- Retries are great for transient errors, but in case of real failures, they are anything but helpful
- Applications should accept that the operation has failed and handle the failure accordingly
- Using HTTP retries carelessly could result in creating a Denial-of-Service attack within your own software

Avælgo

As noted earlier, you should handle faults that might take a variable amount of time to recover from, as might happen when you try to connect to a remote service or resource. Handling this type of fault can improve the stability and resiliency of an application.

In a distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network connections and timeouts, or if resources are responding slowly or are temporarily unavailable. These faults typically correct themselves after a short time, and a robust cloud application should be prepared to handle them by using a strategy like the “Retry pattern”.

However, there can also be situations where faults are due to unanticipated events that might take much longer to fix. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations, it might be pointless for an application to continually retry an operation that’s unlikely to succeed.

Instead, the application should be coded to accept that the operation has failed and handle the failure accordingly.

Using Http retries carelessly could result in creating a Denial of Service (DoS) attack within your own software. As a microservice fails or performs slowly, multiple clients might repeatedly retry failed requests. That creates a dangerous risk of exponentially increasing traffic targeted at the failing service.
Therefore, you need some kind of defense barrier so that excessive requests stop when it isn't worth to keep trying. That defense barrier is precisely the circuit breaker.

Implement Circuit Breaker pattern with HttpClientFactory and Polly

- Circuit Breaker pattern prevents an application from performing an operation that's likely to fail

```
//ConfigureServices()  - Startup.cs
services.AddHttpClient<IBasketService, BasketService>()
        .SetHandlerLifetime(TimeSpan.FromMinutes(5))  //Sample. Default lifetime is 2
minutes
        .AddHttpMessageHandler<HttpClientAuthorizationDelegatingHandler>()
        .AddPolicyHandler(GetRetryPolicy())
        .AddPolicyHandler(GetCircuitBreakerPolicy());
```

```
static IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .CircuitBreakerAsync(5, TimeSpan.FromSeconds(30));
}
```

Avælgo

The Circuit Breaker pattern has a different purpose than the "Retry pattern". The "Retry pattern" enables an application to retry an operation in the expectation that the operation will eventually succeed. The Circuit Breaker pattern prevents an application from performing an operation that's likely to fail. An application can combine these two patterns. However, the retry logic should be sensitive to any exception returned by the circuit breaker, and it should abandon retry attempts if the circuit breaker indicates that a fault is not transient.

As when implementing retries, the recommended approach for circuit breakers is to take advantage of proven .NET libraries like Polly and its native integration with HttpClientFactory.
Adding a circuit breaker policy into your HttpClientFactory outgoing middleware pipeline is as simple as adding a single incremental piece of code to what you already have when using HttpClientFactory.
The only addition here to the code used for HTTP call retries is the code where you add the Circuit Breaker policy to the list of policies to use.

The AddPolicyHandler() method is what adds policies to the HttpClient objects you'll use. In this case, it's adding a Polly policy for a circuit breaker.

To have a more modular approach, the Circuit Breaker Policy is defined in a separate method called GetCircuitBreakerPolicy().

In the code example above, the circuit breaker policy is configured so it breaks or opens the circuit when there have been five consecutive faults when retrying the Http requests. When that happens, the circuit will break for 30 seconds: in that period, calls will be failed immediately by the circuit-breaker rather than actually be placed. The policy automatically interprets relevant exceptions and HTTP status codes as faults. (https://docs.microsoft.com/aspnet/core/fundamentals/http-requests)

Circuit breakers should also be used to redirect requests to a fallback infrastructure if you had issues in a particular resource that's deployed in a different environment than the client application or service that's performing the HTTP call. That way, if there's an outage in the datacenter that impacts only your backend microservices but not your client applications, the client applications can redirect to the fallback services. Polly is planning a new policy to automate this failover policy scenario

Resources

- Circuit Breaker pattern
  - https://docs.microsoft.com/azure/architecture/patterns/circuit-breaker

Microservices-based applications health monitoring

Microservices-based applications health monitoring

- In distributed systems, microservices need to know that the rest of the system environment is health
- It is typical for some hearbeats or signals to be sent out periodically
  - Otherwise, the application might face risks when deployments occur

Avælgo

Microservices-based applications often use heartbeats or health checks to enable their performance monitors, schedulers, and orchestrators to keep track of the multitude of services. If services cannot send some sort of “I’m alive” signal, either on demand or on a schedule, your application might face risks when you deploy updates, or it might just detect failures too late and not be able to stop cascading failures that can end up in major outages.

Implement health checks in ASP.NET Core services

- ASP.NET Core 2.2 offers a built-in health check feature
- Health check services and middleware allow validation if external resources needed for the application is working properly
- Healthy is a custom definition

```
 // Startup.cs from .NET Core 2.2 Web API sample
public void ConfigureServices(IServiceCollection services)
{
  services.AddHealthChecks()
        // Add a health check for a SQL database
        .AddCheck("MyDatabase ", new
SqlConnectionHealthCheck(Configuration["ConnectionStrings:DefaultConnection"]));
}
```

Avaelgo

When developing an ASP.NET Core microservice or web application, you can use the built-in health checks feature that was released in ASP .NET Core 2.2. Like many ASP.NET Core features, health checks come with a set of services and a middleware.

Health check services and middleware are easy to use and provide capabilities that let you validate if any external resource needed for your application (like a SQL Server database or a remote API) is working properly. When you use this feature, you can also decide what it means that the resource is healthy, as we explain later.

In .NET Core 2.2, with the built-in APIs, you can configure the services, add a Health Check for the microservice and its dependent SQL Server database.
In the previous code, services.AddHealthChecks() method configures a basic HTTP check that returns a Status code **200** with “Healthy”. Further, AddCheck() extension method configures a custom SqlConnectionHealthCheck that checks the related SQL Database’s health. The AddCheck() method adds a new health check with a specified name and the implementation of type IHealthCheck. You can add multiple Health Checks using AddCheck method, so a microservice will not provide a “healthy” status until all its checks are healthy.

Custom health check

```
public class SqlConnectionHealthCheck : IHealthCheck
{
    private static readonly string DefaultTestQuery = "Select 1";
    public string ConnectionString { get; }
    public string TestQuery { get; }

    public SqlConnectionHealthCheck(string connectionString) : this(connectionString,
testQuery: DefaultTestQuery) { }

    public SqlConnectionHealthCheck(string connectionString, string testQuery)
    {
        ConnectionString = connectionString ?? throw new
ArgumentNullException(nameof(connectionString));
        TestQuery = testQuery;
    }
    public async Task<HealthCheckResult> CheckHealthAsync(HealthCheckContext context,
CancellationToken cancellationToken = default(CancellationToken))
    {
        using (var connection = new SqlConnection(ConnectionString))
        {
            try
            {
                await connection.OpenAsync(cancellationToken);
                if (TestQuery != null)
                {
                    var command = connection.CreateCommand();
                    command.CommandText = TestQuery;
                    await command.ExecuteNonQueryAsync(cancellationToken);
                }
            }
            catch (DbException ex)
            {
                return new HealthCheckResult(status:
context.Registration.FailureStatus, exception: ex);
            }
        }
        return HealthCheckResult.Healthy();
    }
}
```

Avælgo

SqlConnectionHealthCheck is a custom class that implements IHealthCheck, which takes a connection string as a constructor parameter and executes a simple query to check if the connection to the SQL database is successful. It returns HealthCheckResult.Healthy() if the query was executed successfully and a FailureStatus with the actual exception when it fails

Health check middleware

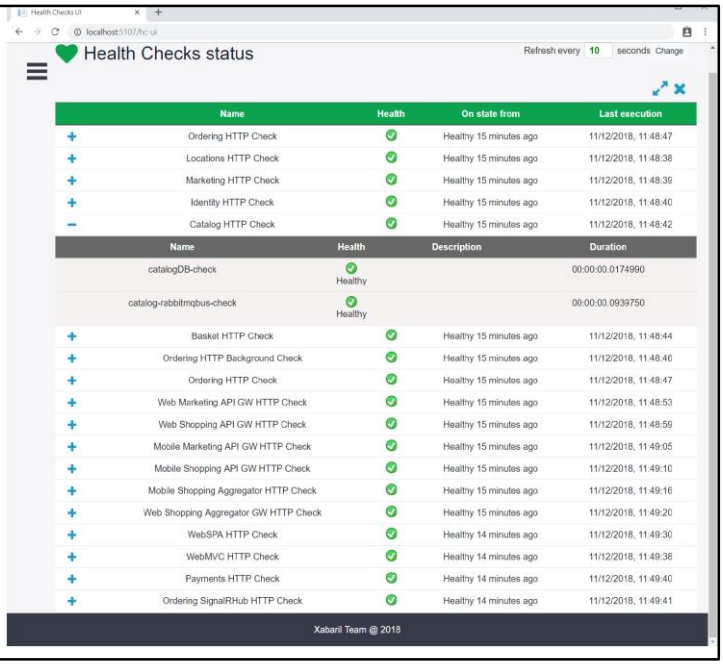
```
// Startup.cs from .NET Core 2.2 Web Api sample
//
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
…
    app.UseHealthChecks("/hc");
…
}
```

Avælgo

Finally, we create a middleware that responds to the url path “/hc”:
When the endpoint *<yourmicroservice>/hc* is invoked, it runs all the health checks that are configured in the AddHealthChecks() method in the Startup class and shows the result.

Use Watchdogs

- Watchdogs are separate services that watch health and load across services and report health about the microservices by querying with the HealthChecks library
- AspNetCore.Diagnostics.HealthChecks provides a UI NuGet package



Avælgo

A watchdog is a separate service that can watch health and load across services, and report health about the microservices by querying with the HealthChecks library introduced earlier. This can help prevent errors that would not be detected based on the view of a single service. Watchdogs also are a good place to host code that can perform remediation actions for known conditions without user interaction.

Resources

- HealthChecks and HealthChecks UI for ASP.NET Core
  - https://github.com/Xabaril/AspNetCore.Diagnostics.HealthChecks
- Introduction to Service Fabric health monitoring
  - https://docs.microsoft.com/azure/service-fabric/service-fabric-health-introduction
- Azure Application Insights
  - https://azure.microsoft.com/services/application-insights/
- Microsoft Operations Management Suite
  - https://www.microsoft.com/cloud-platform/operations-management-suite

Avælgo