Key Characteristics of Microservices

1. **Domain-Driven Design:** Functional decomposition can be easily achieved using Eric Evans's DDD approach.
2. **Single Responsibility Principle:** Each service is responsible for a single part of the functionality, and does it well.
3. **Explicitly Published Interface:** A producer service publishes an interface that is used by a consumer service.
4. **Independent DURS (Deploy, Update, Replace, Scale):** Each service can be independently deployed, updated, replaced, and scaled.
5. **Lightweight Communication:** REST over HTTP, STOMP over WebSocket, and other similar lightweight protocols are used for communication between services.

Benefits of Microservices

1. **Independent scaling:** Each microservice can scale independently via X-axis scaling (cloning with more CPU or memory) and Z-axis scaling (sharding), based upon their needs. This is very different from monolithic applications, which may have very different requirements that must be deployed together.
2. **Independent upgrades:** Each service can be deployed independent of other services. Any change local to a service can be easily made by a developer without requiring coordination with other teams. For example, performance of a service can be improved by changing the underlying implementation. As a result this maintains the agility of the microservice. This is also a great enabler of CI/CD.
3. **Easy maintenance:** Code in a microservice is restricted to one function and is thus easier to understand. IDEs can load the smaller amounts of code more easily, and increased readability can keep developers more productive.
4. **Potential heterogeneity and polyglotism:** Developers are free to pick the language and stack that are best suited for their service. This enables one to rewrite the service using better languages and technologies as opposed to being penalized because of past decisions, and gives freedom of choice when picking a technology, tool, or framework.
5. **Fault and resource isolation:** A misbehaving service, such as a memory leak or an unclosed database connection, will only affect that service, as opposed to an entire monolithic application. This improves fault isolation and limits how much of an application a failure can affect.
6. **Improved communication across teams:** A microservice is typically built by a full-stack team. All members related to a domain work together in a single team, which significantly improves communication between team members, as they share the same end goal.

Operational Requirements for Microservices

Microservices are not the silver bullet that will solve all architectural problems in your applications. Implementing microservices may help, but that is often just the byproduct of refactoring your application and typically rewriting code using guidelines required by this architecture style. True success requires significant investment.

1. **Service Replication:** Each service needs to replicate, typically using X-axis cloning or Y-axis partitioning. There should be a standard mechanism by which services can easily scale based upon metadata. A PaaS, such as OpenShift by Red Hat, can simplify this functionality.
2. **Service Discovery:** In a microservice world, multiple services are typically distributed in a PaaS

environment. Immutable infrastructure is provided by containers or immutable VM images. Services may scale up and down based upon certain pre-defined metrics. The exact address of a service may not be known until the service is deployed and ready to be used.

The dynamic nature of a service's endpoint address is handled by service registration and discovery. Each service registers with a broker and provides more details about itself (including the endpoint address). Other consumer services then query the broker to find out the location of a service and invoke it. There are several ways to register and query services such as ZooKeeper, etcd, consul, Kubernetes, Netflix Eureka, and others.

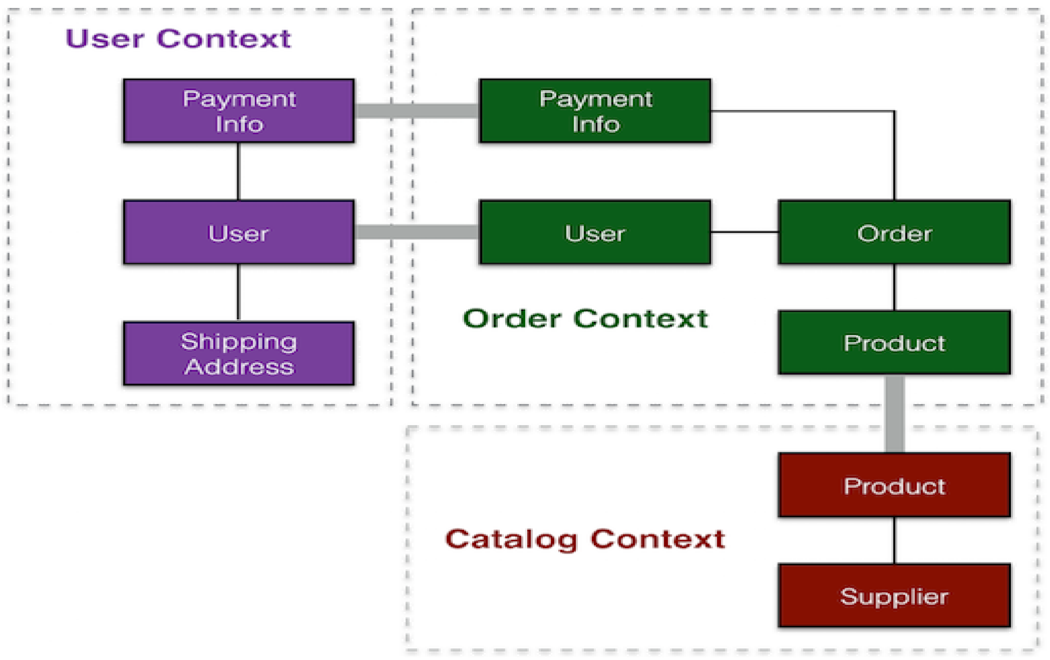
3. **Service Monitoring:** One of the most important aspects of a distributed system is service monitoring and logging. This enables one to take proactive action if, for example, a service is consuming unexpected resources. The ELK Stack can aggregate logs from different microservices, provide a consistent visualization over them, and make that data available to business users. Other possible tools for distributed logging are Syslog, Logentries, and Loggly.
4. **Resiliency:** Software failure will occur, no matter how much and how hard you test. This is all the more important when multiple microservices are distributed all over the Internet. The key concern is not "how to avoid failure" but "how to deal with failure." It's important for services to automatically take corrective action to ensure user experience is not impacted. The Circuit Breaker pattern allows one to build resiliency in software—Netflix's Hystrix and Ribbon are good libraries that implement this pattern.
5. **DevOps:** Continuous Integration and Continuous Deployment (CI/CD) are very important in order for microservices-based applications to succeed. These practices are required so that errors are identified early, and so little to no coordination is required between different teams building different microservices.

Patterns in Microservices

1. Use Bounded Contexts to Identify Candidates for Microservices

Bounded contexts (a pattern from domain-driven design) are a great way to identify what parts of the domain model of a monolithic application can be decomposed into microservices. In an ideal decomposition, every bounded context can be extracted out as a separate microservice, which communicates with other similarly-extracted microservices through well defined APIs. It is not necessary to share the model classes of a microservice with consumers; the model objects should be hidden as you wish to minimize binary dependencies between your microservices.

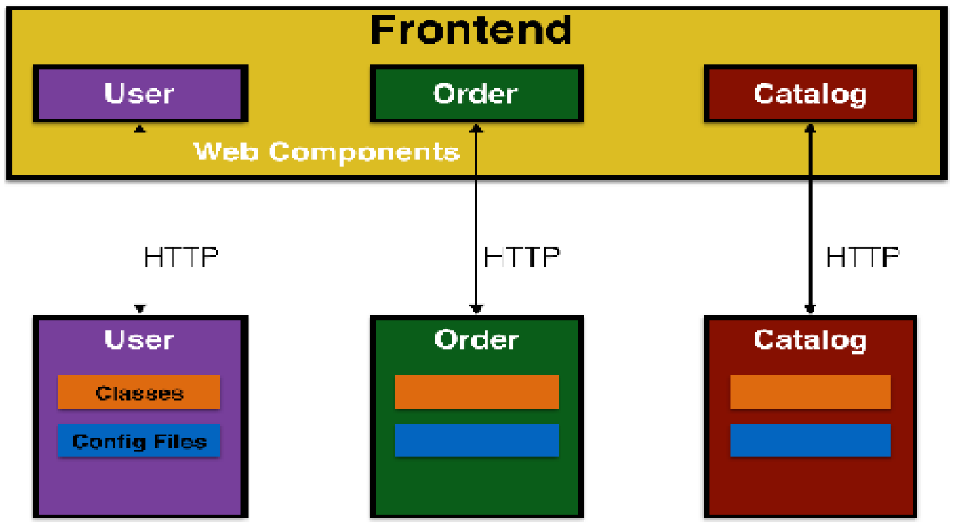
The use of an anti-corruption layer is strongly recommended during the process of decomposing a monolith. The anti-corruption layer allows the model in one microservice to change without impacting the consumers of the microservice.



2. Designing Frontends for Microservices

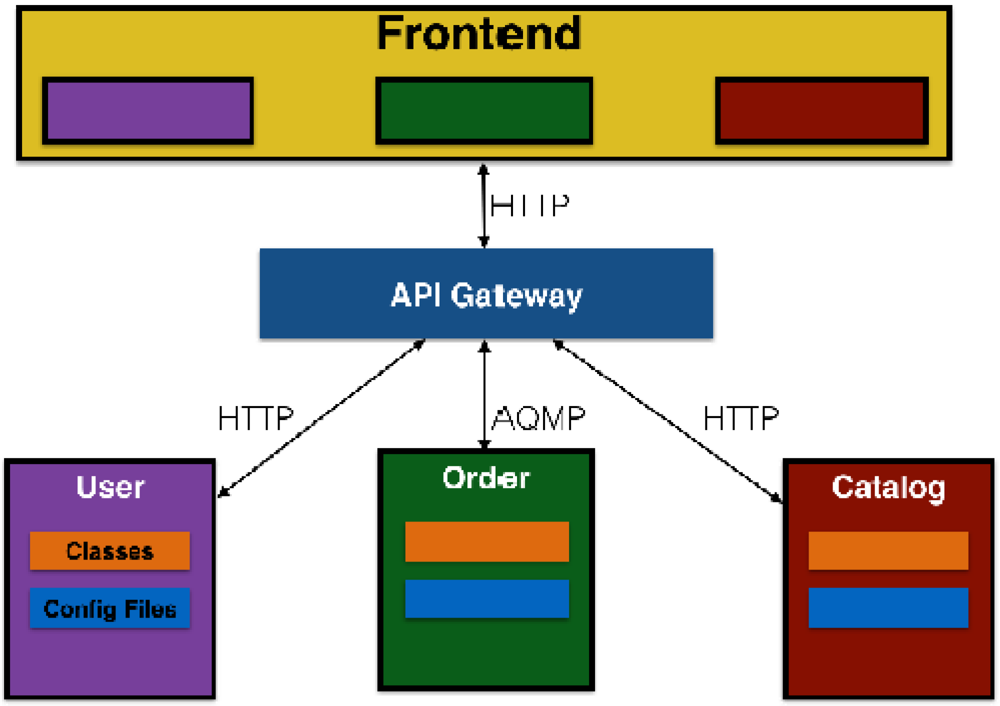
In a typical monolith, one team is responsible for developing and maintaining the frontend. With a microservices-architecture, multiple teams would be involved, and this could easily grow into an unmanageable problem.

This could be resolved by componentizing various part of the frontend so that each microservice team can develop in a relatively isolated manner. Each microservice team will develop and maintain it's own set of components. Changes to various parts of the frontend are encapsulated to components and thus do not leak into other parts of the frontend.



3. Use an API Gateway to Centralize Access to Microservices
The frontend, acting as a client of microservices, may fetch data from several different microservices. Some of these microservices may not be capable of responding to protocols native to the web (HTTP+JSON/XML), thus requiring translation of messages from one protocol into another. Concerns like authentication and authorization, request-response translation among others can be handled in a centralized manner in a facade.

Consider using an API Gateway, which acts as a facade that centralizes the aforementioned concerns at the network perimeter. Obviously, the API Gateway would respond to client requests over a protocol native to the web, while communicating with underlying microservices using the approach preferred by the microservices. Clients of microservices may identify themselves to the API Gateway through a token-authentication scheme like OAuth. The token may be revalidated again in downstream microservices to enforce defense in depth.



4. Database Design and Refactorings
Unlike a monolithic application, which can be designed to use a single database, microservices should be designed to use a separate logical database for each microservice. Sharing databases is discouraged and is an anti-pattern, as it forces one or more teams to wait for others, in the event of database schema updates; the database essentially becomes an API between the teams. One can use database schemas or other logical schemes to create a separate namespace of database objects, to avoid the need to create multiple physical databases.

Additionally, one should consider the use of a database migration tool like Liquibase or Flyway when migrating from a monolithic architecture to a microservices architecture. The single monolithic database would have to be cloned to every microservice, and migrations have to be applied on every database to transform and extract the data appropriate to the microservice owning the database. Not every object in the original database of the monolith would be of interest to the individual microservices.

5. Packaging and Deploying Services
Each microservice can in theory utilize its own technology stack, to enable the team to develop and maintain it in isolation with few external dependencies. It is quite possible for a single organization to utilize multiple runtime platforms (Java/JVM, Node, .NET, etc.) in a microservices architecture. This potentially opens up problems with provisioning these various runtime platforms on the datacenter hosts/nodes.

Consider the use of virtualization technologies that allow the tech stacks to be packaged and deployed in separate virtual environments. To take this further, consider the use of containers as a lightweight-virtualization technology that allows the microservice and its runtime environment to be packaged up in a single image. With containers, one forgoes the need for packaging and running a guest OS on the host OS. Following which, consider the use of a container orchestration technology like Kubernetes, which allows you to define the various deployments in a manifest file, or a container platform, such as OpenShift. The manifest defines the intended state of the datacenter with various containers and associated policies, enabling a DevOps culture to spread and grow across development and operations teams.

6. Event-driven architecture
Microservices architectures are renowned for being eventually consistent, given that there are multiple datastores that store state within the architecture. Individual microservices can themselves be strongly consistent, but the system as a whole may exhibit eventual consistency in parts. To account for the eventual consistency property of a microservices architecture, one should consider the use of an event-driven architecture where data-changes in one microservice are relayed to interested microservices through events.

A pub-sub messaging architecture may be employed to realize the event-driven architecture. One microservice may publish events as they occur in its context, and the events would be communicated to all interested microservices, that can proceed to update their own state. Events are a means to transfer state from one service to another.

7. Consumer-Driven Contract Testing
One may start off using integration testing to verify the correctness of changes performed during development. This is, however, an expensive process, since it requires booting up a microservice, the microservice's dependencies, and all the microservice's consumers, to verify that changes in the microservice have not broken the expectations of the consumers.

This is where consumer-driven contracts aid in bringing in agility. With a consumer-driven contract, a consumer will declare a contract on what it expects from a provider.

Introduction Twelve Factor App

In the modern era, software is commonly delivered as a service: called web apps, or software-as-a-service. The twelve-factor app is a methodology for building software-as-a-service apps that:

- Use declarative formats for setup automation, to minimize time and cost for new developers joining the project;
- Have a clean contract with the underlying operating system, offering maximum portability between execution environments;
- Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration;
- Minimize divergence between development and production, enabling continuous deployment for maximum agility;
- And can scale up without significant changes to tooling, architecture, or development practices.

The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

Background
The contributors to this document have been directly involved in the development and deployment of hundreds of apps, and indirectly witnessed the development, operation, and scaling of hundreds of thousands of apps via our work on the Heroku platform.

This document synthesizes all of our experience and observations on a wide variety of software-as-a-service apps in the wild. It is a triangulation on ideal practices for app development, paying particular attention to the dynamics of the organic growth of an app over time, the dynamics of collaboration between developers working on the app's codebase, and avoiding the cost of software erosion.

Our motivation is to raise awareness of some systemic problems we've seen in modern application development, to provide a shared vocabulary for discussing those problems, and to offer a set of broad conceptual solutions to those problems with accompanying terminology. The format is inspired by Martin Fowler's books Patterns of Enterprise Application Architecture and Refactoring.

Who should read this document?
Any developer building applications which run as a service. Ops engineers who deploy or manage such applications.

The Twelve Factors
I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing services

Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

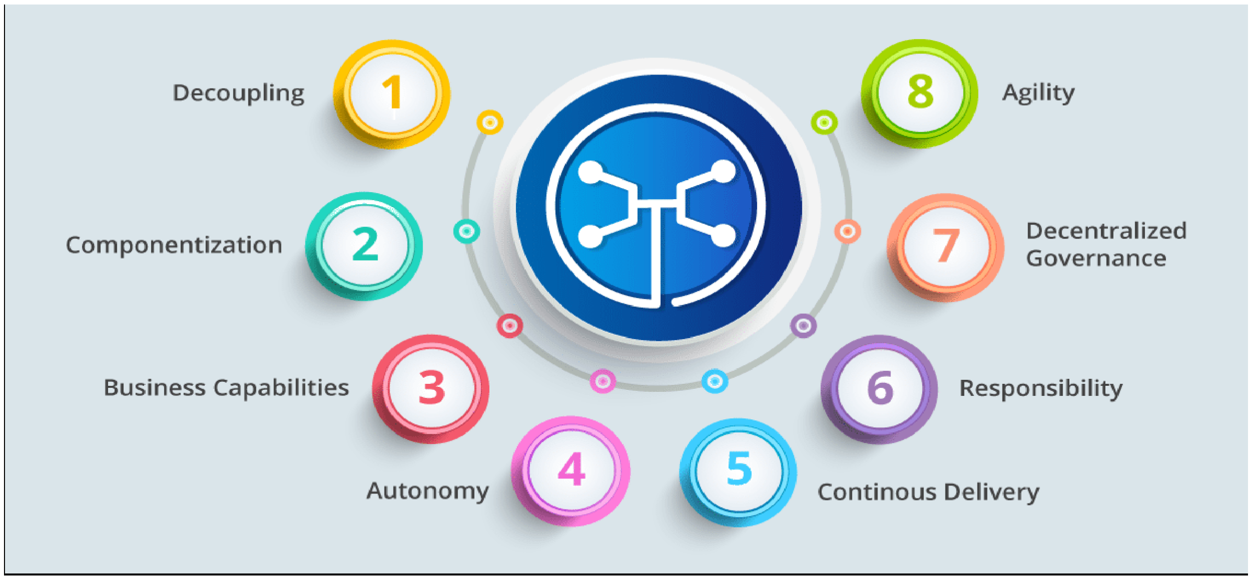
VIII. Concurrency
Scale out via the process model

IX. Disposability
Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity
Keep development, staging, and production as similar as possible

XI. Logs
Treat logs as event streams

XII. Admin processes
Run admin/management tasks as one-off processes



1. **Decoupling** – Services within a system are largely decoupled. So the application as a whole can be easily built, altered, and scaled
2. **Componentization** – Microservices are treated as independent components that can be easily re-placed and upgraded
3. **Business Capabilities** – Microservices are very simple and focus on a single capability
4. **Autonomy** – Developers and teams can work independently of each other, thus increasing speed

5. **Continous Delivery** – Allows frequent releases of software, through systematic automation of software creation, testing, and approval
6. **Responsibility** – Microservices do not focus on applications as projects. Instead, they treat applications as products for which they are responsible
7. **Decentralized Governance** – The focus is on using the right tool for the right job. That means there is no standardized pattern or any technology pattern. Developers have the freedom to choose the best useful tools to solve their problems
8. **Agility** – Microservices support agile development. Any new feature can be quickly developed and discarded again

Advantages Of Microservices

1. **Independent Development** – All microservices can be easily developed based on their individual functionality
2. **Independent Deployment** – Based on their services, they can be individually deployed in any application
3. **Fault Isolation** – Even if one service of the application does not work, the system still continues to function
4. **Mixed Technology Stack** – Different languages and technologies can be used to build different services of the same application
5. **Granular Scaling** – Individual components can scale as per need, there is no need to scale all components together

Conway's law

"organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations."

The law is based on the reasoning that in order for a software module to function, multiple authors must communicate frequently with each other. Therefore, the software interface structure of a system will reflect the social boundaries of the organization(s) that produced it, across which communication is more difficult. Conway's law was intended as a valid sociological observation, although sometimes it's used in a humorous context. It was dubbed Conway's law by participants at the 1968 National Symposium on Modular Programming.[3]