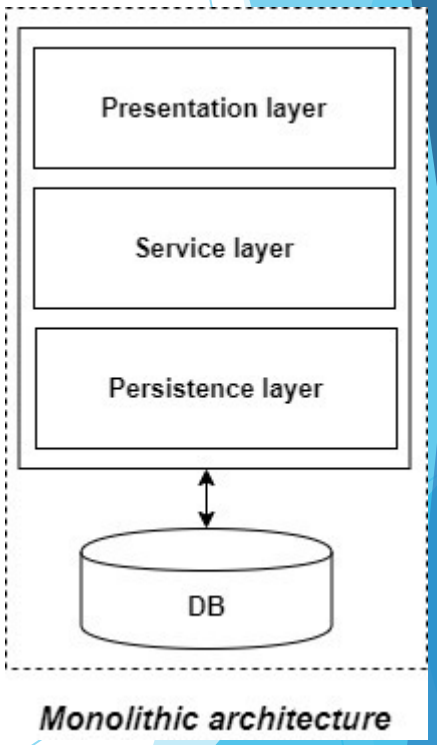
Spring Microservices

Spring Boot

Monolithic applications

- If all the functionalities of a project exist in a single codebase, then that application is known as a monolithic application.
- We all must have designed a monolithic application in our lives in which we were given a problem statement and were asked to design a system with various functionalities.
- We design our application in various layers like presentation, service, and persistence and then deploy that codebase as a single jar/war file.
- This is nothing but a monolithic application, where “mono” represents the single codebase containing all the required functionalities.



Monolithic architecture

Advantages of a monolithic architecture

- **Easy deployment** – One executable file or directory makes deployment easier.
- **Development** – When an application is built with one code base, it is easier to develop.
- **Performance** – In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices.
- **Simplified testing** – Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application.

  Easy debugging – With all code located in one place, it's easier to follow a request and find an issue.

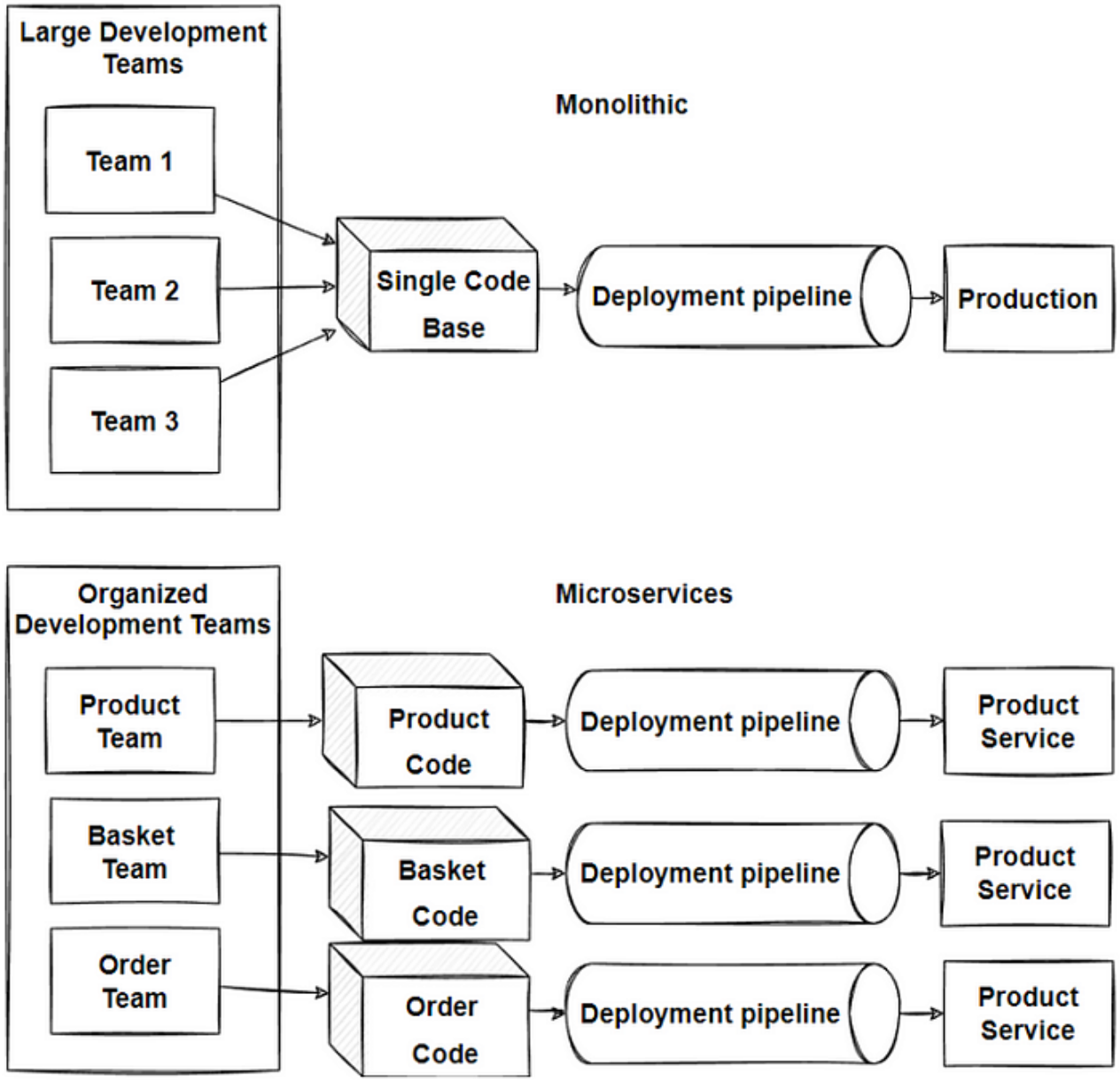
Challenges of Monolithic Architecture

- **Inflexible** - Monolithic applications cannot be built using different technologies
- **Unreliable** - Even if one feature of the system does not work, then the entire system does not work
- **Unscalable** - Applications cannot be scaled easily since each time the application needs to be updated, the complete system has to be rebuilt
- **Blocks Continous Development** - Many features of the applications cannot be built and deployed at the same time
- **Slow Development** - Development in monolithic applications take lot of time to be built since each and every feature has to be built one after the other
- **Not Fit For Complex Applications** - Features of complex applications have tightly coupled dependencies

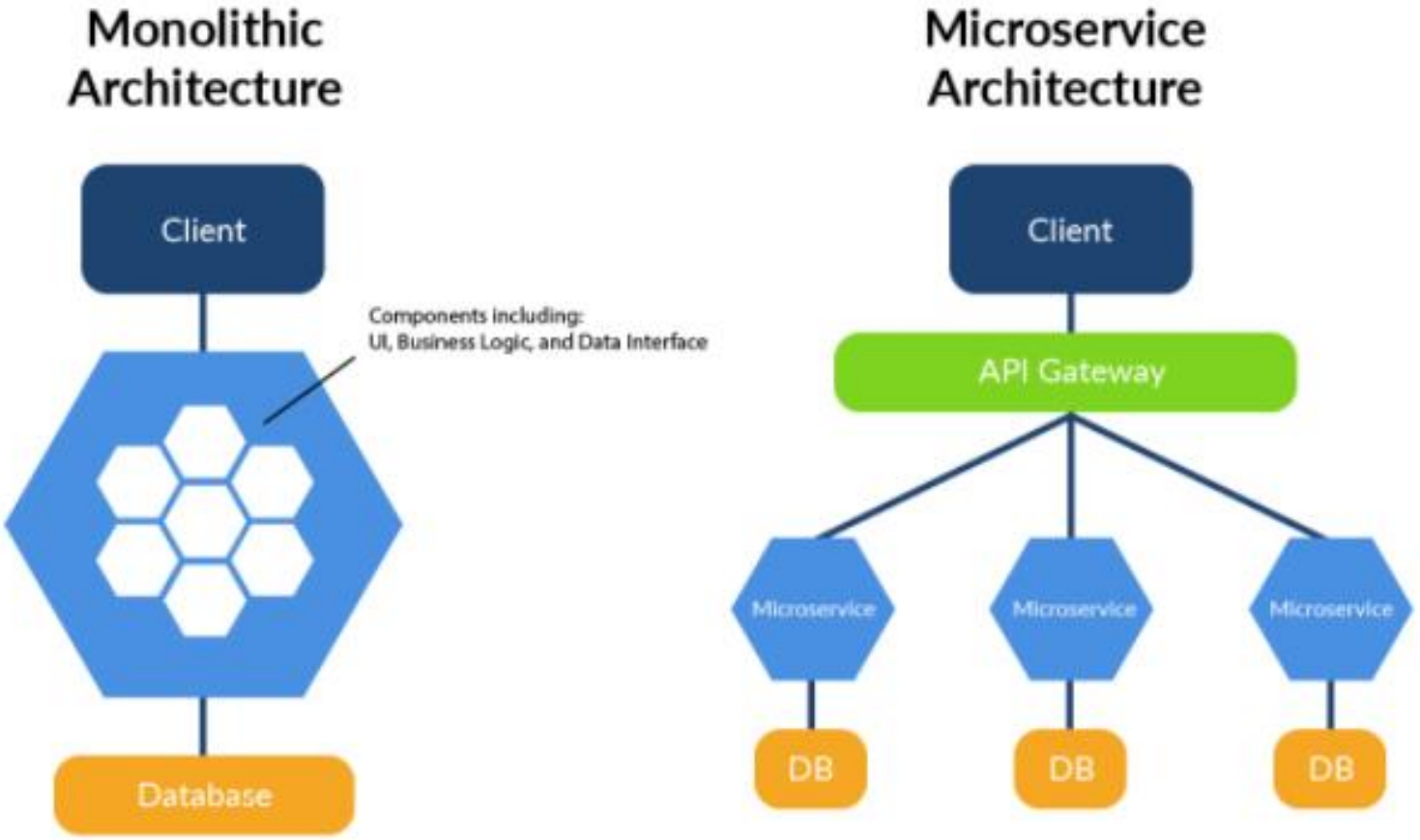
Microservices

- Definition: According to Sam Newman, "Microservices are the small services that work together."
- Server ties all the small services together in form of a single application: Microservice.
- According to James Lewis and Martin Fowler, "The microservice architectural style is an approach to develop a single application as a suite of small services. Each microservice runs its process and communicates with lightweight mechanisms. These services are built around business capabilities and independently developed by fully automated deployment machinery."
- There is a bare minimum of centralized management of these services, which may be written in different programming language and use different data storage technologies.
- Any Service is not responsible for managing other services. Only Server will serve us with microservices.
- All Microservice Servers are cloud-servers: they take microservices, run it as cloud-enabled RESTful services.
- Points to remember
  - These are the services which are exposed by REST.
  - These are small well-chosen deployable units.
  - The services must be cloud-enabled.
- The microservice defines an approach to the architecture that divides an application into a pool of loosely coupled services that implements business requirements. It is next to Service-Oriented Architecture (SOA). The most important feature of the microservice-based architecture is that it can perform continuous delivery of a large and complex application.
- Microservice helps in breaking the application and build a logically independent smaller applications. For example, we can build a cloud application with the help of Amazon AWS with minimum efforts.

Architecture Comparison: Monolithic vs Microservices



Simple Comparison: Architecture



Principles of Microservices

- Single Responsibility Principle
  - The single responsibility principle states that a class or a module in a program should have only one responsibility. Any microservice cannot serve more than one responsibility, at a time.
- Modeled around business domain
  - Microservice never restrict itself from accepting appropriate technology stack or database. The stack or database is most suitable for solving the business purpose.
- Isolated Failure
  - The large application can remain mostly unaffected by the failure of a single module. It is possible that a service can fail at any time. So, it is important to detect failure quickly, if possible, automatically restore failure.
- Infrastructure Automation
  - The infrastructure automation is the process of scripting environments. With the help of scripting environment, we can apply the same configuration to a single node or thousands of nodes. It is also known as configuration management, scripted infrastructures, and system configuration management.
- Deploy independently
  - Microservices are platform agnostic. It means we can design and deploy them independently without affecting the other services.

Features of Microservices

Features of Microservices

- **Decoupling** – Services within a system are largely decoupled. So the application as a whole can be easily built, altered, and scaled
- **Componentization** – Microservices are treated as independent components that can be easily replaced and upgraded
- **Business Capabilities** – Microservices are very simple and focus on a single capability
- **Autonomy** – Developers and teams can work independently of each other, thus increasing speed
- **Continous Delivery** – Allows frequent releases of software, through systematic automation of software creation, testing, and approval
- **Responsibility** – Microservices do not focus on applications as projects. Instead, they treat applications as products for which they are responsible
- **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
- **Agility** – Any new feature can be quickly developed and discarded again

Advantages of Microservices

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

Disadvantages of Microservices

- Microservices has all the associated complexities of the distributed system.
- There is a higher chance of failure during communication between different services.
- Difficult to manage a large number of services.
- The developer needs to solve the problem, such as network latency and load balancing.
- Complex testing over a distributed environment.

Challenges of Microservices Architecture

- Microservice architecture is more complex than the legacy system. The microservice environment becomes more complicated because the team has to manage and support many moving parts. Here are some of the top challenges that an organization face in their microservices journey:
- Bounded Context: The bounded context concept originated in Domain-Driven Design (DDD) circles. A bounded context clarifies, encapsulates, and defines the specific responsibility to the model. It ensures that the domain will not be distracted from the outside.
- Dynamic Scale up and Scale Down: The loads on the different microservices may be at a different instance of the type. As well as auto-scaling up your microservice should auto-scale down.
- Monitoring: The traditional way of monitoring will not align well with microservices because we have multiple services making up the same functionality previously supported by a single application.
- Fault Tolerance: Fault tolerance is the individual service that does not bring down the overall system. The application can operate at a certain degree of satisfaction when the failure occurs.
- Cyclic dependencies: Dependency management across different services, and its functionality is very important. The cyclic dependency can create a problem, if not identified and resolved promptly.
- DevOps Culture: Microservices fits perfectly into the DevOps. It provides faster delivery service, visibility across data, and cost-effective data. It can extend their use of containerization switch from Service-Oriented-Architecture (SOA) to Microservice Architecture (MSA).

MSA Vs SOA

Microservice Based Architecture (MSA)	Service-Oriented Architecture (SOA)
Microservices uses **lightweight protocols** such as **REST**, and **HTTP**, etc.	SOA supports **multi-message protocols**.
It focuses on **decoupling**.	It focuses on application service **reusability**.
It uses a **simple messaging system** for communication.	It uses **Enterprise Service Bus** (ESB) for communication.
Microservices follows "**share as little as possible**" architecture approach.	SOA follows "**share as much as possible architecture**" approach.
Microservices are much better in **fault tolerance** in comparison to SOA.	SOA is not better in fault tolerance in comparison to MSA.
Each microservice have an **independent** database.	SOA services share the **whole** data storage.
MSA used **modern** relational databases.	SOA used **traditional** relational databases.
MSA tries to **minimize** sharing through bounded context (the coupling of components and its data as a single unit with minimal dependencies).	SOA **enhances** component sharing.
It is better suited for the **smaller** and **well portioned**, web-based system.	It is better for a **large** and **complex** business application environment.

Microservices vs. monolithic ? Which one to choose

- A monolithic application is built as a single unified unit while a microservices architecture is a collection of smaller, independently deployable services.
- Which one is right for you? It depends on a number of factors.
- Microservices may not be for everyone.
- A legacy monolith may work perfectly well, and breaking it down may not be worth the trouble.
- But as organizations grow and the demands on their applications increase, microservices architecture can be worthwhile.

Components of Microservices

- Spring Cloud Config Server
  - Spring Cloud Config Server provides the HTTP resource-based API for external configuration in the distributed system. We can enable the Spring Cloud Config Server by using the annotation **@EnableConfigServer**.
- Netflix Eureka Naming Server
  - Netflix Eureka Server is a discovery server. It provides the REST interface to the outside for communicating with it. A microservice after coming up, register itself as a discovery client. The Eureka server also has another software module called **Eureka Client**. Eureka client interacts with the Eureka server for service discovery. The Eureka client also balances the client requests.
- Hystrix Server
  - Hystrix server acts as a fault-tolerance robust system. It is used to avoid complete failure of an application. It does this by using the **Circuit Breaker mechanism**. If the application is running without any issue, the circuit remains closed. If there is an error encountered in the application, the Hystrix Server opens the circuit. The Hystrix server stops the further request to calling service. It provides a highly robust system.
- Netflix Zuul API Gateway Server
  - Netflix Zuul Server is a gateway server from where all the client request has passed through. It acts as a unified interface to a client. It also has an inbuilt load balancer to load the balance of all incoming request from the client.
- Netflix Ribbon
  - Netflix Ribbon is the client-side Inter-Process Communication (IPC) library. It provides the client-side balancing algorithm. It uses a Round Robin Load Balancing: Load balancing, Fault tolerance, Multiple protocols(HTTP, TCP, UDP), Caching and Batching
- Zipkin Distributed Server
  - Zipkin is an open-source project m project. That provides a mechanism for sending, receiving, and visualization traces.

Port number

Application	Port
Spring Cloud Config Server	8888
Netflix Eureka Naming Server	8761
Netflix Zuul API gateway Server	8765
Zipkin distributed Tracing Server	9411

Companies using Microservices

SpringBoot Microservices Patterns

SB

Microservices: key patterns and considerations

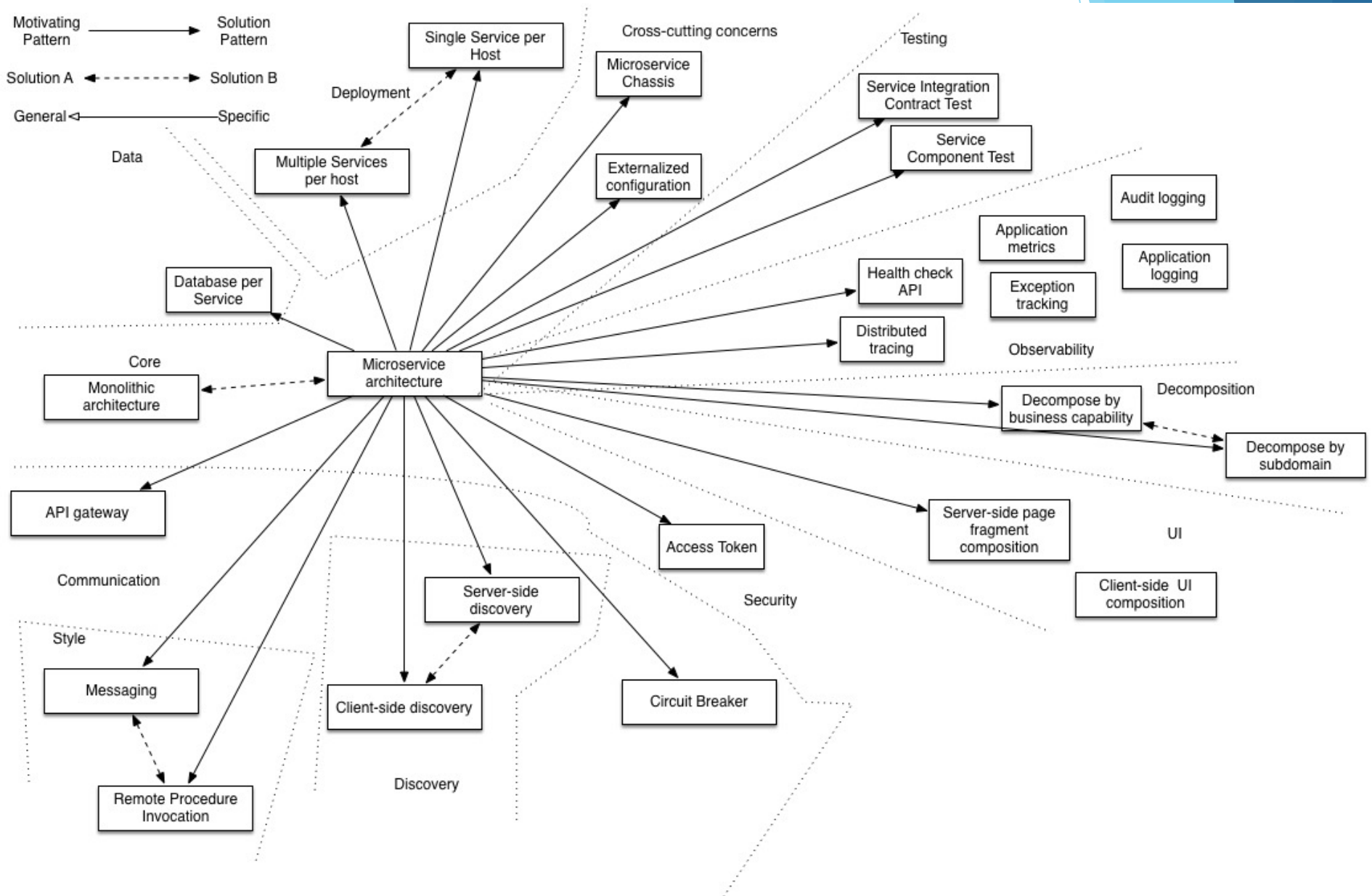
- Spring Boot has become a popular framework for building microservices due to its simplicity and powerful features. When developing microservices with Spring Boot, several design patterns and best practices are commonly employed to ensure scalability, resilience, and maintainability. Here are some key patterns and considerations:
- **Service Registration and Discovery**: Use of a service registry (like Netflix Eureka, Consul, or Spring Cloud Netflix) for service registration and a discovery client to locate services dynamically.
- **Externalized Configuration**: Utilize Spring Boot's externalized configuration capabilities (via application.properties or YAML files) to separate configuration from code, making microservices easier to configure and deploy across different environments.
- **Centralized Logging**: Implement centralized logging using tools like ELK Stack (Elasticsearch, Logstash, Kibana) or Splunk for easier debugging and monitoring of microservices.
- **Database per Service**: Each microservice should ideally have its own database to ensure loose coupling and independence. Use technologies like Spring Data JPA for easy database access and management.
- **API Gateway**: Implement an API Gateway (e.g., Spring Cloud Gateway, Netflix Zuul) to handle routing, load balancing, authentication, and monitoring of requests from clients to microservices.
- **Circuit Breaker**: Implement the Circuit Breaker pattern (e.g., Netflix Hystrix or Resilience4j) to prevent cascading failures and provide fallback options when microservices are unavailable.
- **Event-driven Architecture**: Use asynchronous messaging and event-driven patterns (e.g., Kafka, RabbitMQ, Spring Cloud Stream) to enable communication between microservices while ensuring loose coupling and scalability.
- **Containerization and Orchestration**: Containerize microservices using Docker and orchestrate them with Kubernetes or Docker Swarm for easier deployment, scaling, and management of containers.
- **Fault Tolerance and Resilience**: Design microservices to be resilient to failures by implementing retry mechanisms, timeouts, and fallbacks in critical operations.
- **Monitoring and Health Checks**: Implement health checks and metrics endpoints (e.g., Spring Boot Actuator) to monitor the health and performance of microservices in real-time.
- These patterns help address common challenges in building and deploying microservices, ensuring they are scalable, resilient, and easy to manage in a distributed environment.

Related Design patterns

- There are many patterns related to the Microservices architecture pattern.
- The Monolithic architecture is an alternative to the microservice architecture.
- The other patterns in the Microservice architecture architecture pattern address issues that you will encounter when applying this pattern.

- https://microservices.io/patterns/microservices.html

Related Design patterns



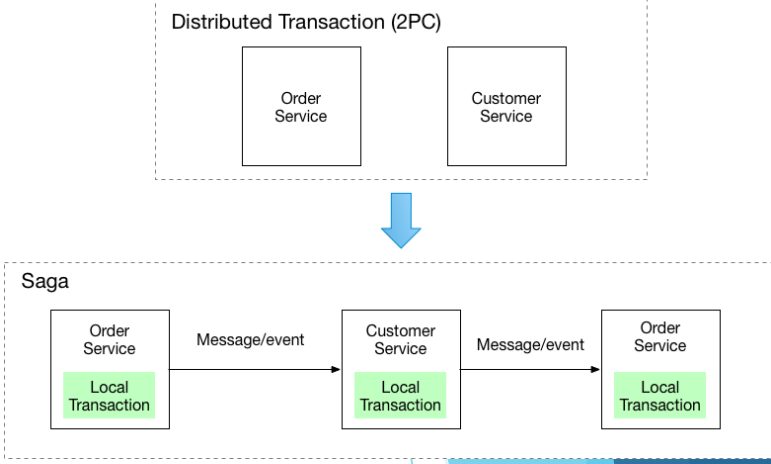
..Related Design patterns

- Service collaboration patterns:
  - Saga, which implements a distributed command as a series of local transactions
  - Command-side replica, which replicas read-only data to the service that implements a command
  - API composition, which implements a distributed query as a series of local queries
  - CQRS, which implements a distributed query as a series of local queries
- The Messaging and Remote Procedure Invocation patterns are two different ways that services can communicate.
- The Database per Service pattern describes how each service has its own database in order to ensure loose coupling.
- The API Gateway pattern defines how clients access the services in a microservice architecture.
- The Client-side Discovery and Server-side Discovery patterns are used to route requests for a client to an available service instance in a microservice architecture.
- Testing patterns: Service Component Test and Service Integration Contract Test
- Circuit Breaker
- Access Token

..Related Design patterns

- Observability patterns:
  - Log aggregation
  - Application metrics
  - Audit logging
  - Distributed tracing
  - Exception tracking
  - Health check API
  - Log deployments and changes
- UI patterns:
  - Server-side page fragment composition
  - Client-side UI composition
- The Single Service per Host and Multiple Services per Host patterns are two different deployment strategies.
- Cross-cutting concerns patterns: Microservice chassis pattern and Externalized configuration

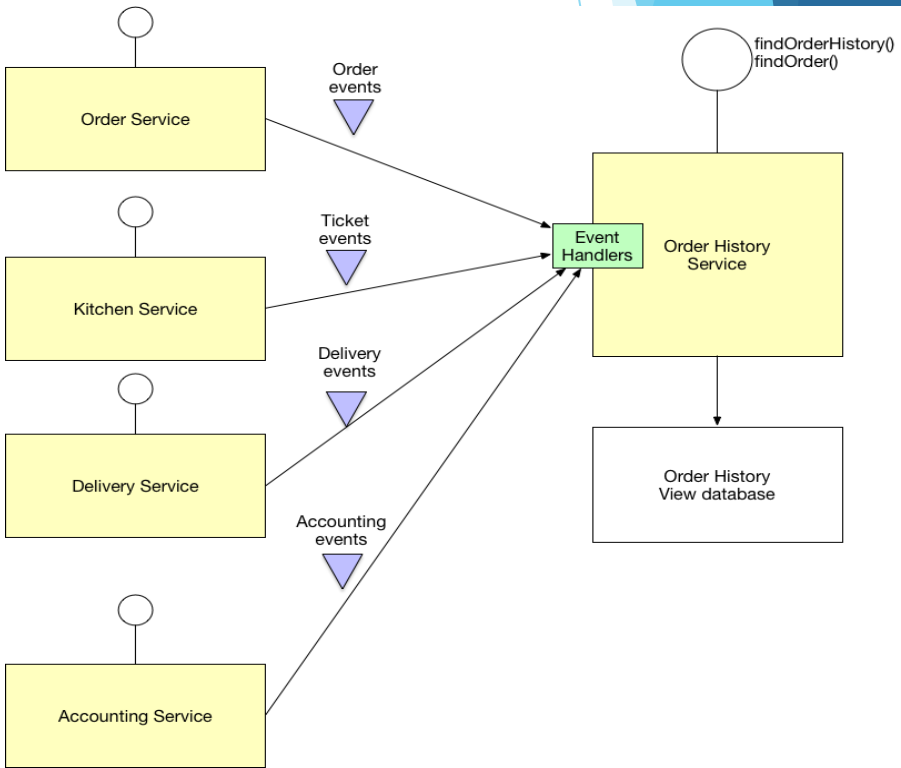
Pattern: Saga



- Each service has its own database.
- Some business transactions, however, span multiple service so you need a mechanism to implement transactions that span services.
- For example, let's imagine that you are building an e-commerce store where customers have a credit limit. The application must ensure that a new order will not exceed the customer's credit limit. Since Orders and Customers are in different databases owned by different services the application cannot simply use a local ACID transaction.
- The Solution is to implement each business transaction that spans multiple services as a saga.
- A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga.
- If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.
- Benefits: It enables an application to maintain data consistency across multiple services without using distributed transactions
- Drawbacks: The programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga.

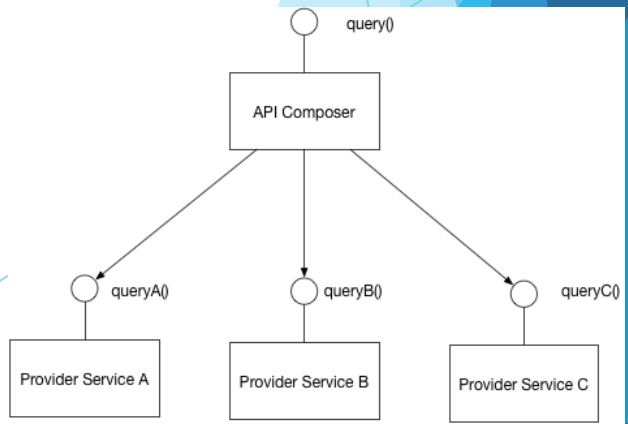
Pattern: Command Query Responsibility Segregation (CQRS)

- CQRS implement a query that retrieves data from multiple services in a microservice architecture.
- Here we define a view database, which is a read-only replica that is designed to support that query. The application keeps the replica up to data by subscribing to Domain events published by the service that own the data.
- Benefits:
  - Supports multiple denormalized views that are scalable and performant
  - Improved separation of concerns = simpler command and query models
  - Necessary in an event sourced architecture
- Drawbacks:
  - Increased complexity
  - Potential code duplication
  - Replication lag/eventually consistent views



Pattern: API Composition

- Here we implement a query by defining an *API Composer*, which invoking the services that own the data and performs an in-memory join of the results.
- Benefits:
  - It a simple way to query data in a microservice architecture
- Drawbacks:
  - Some queries would result in inefficient, in-memory joins of large datasets.



Pattern: Messaging

- Use asynchronous messaging for inter-service communication. Services communicating by exchanging messages over messaging channels.
- There are several different styles of asynchronous communication:
  - Request/response - a service sends a request message to a recipient and expects to receive a reply message promptly
  - Notifications - a sender sends a message a recipient but does not expect a reply. Nor is one sent.
  - Request/asynchronous response - a service sends a request message to a recipient and expects to receive a reply message eventually
  - Publish/subscribe - a service publishes a message to zero or more recipients
  - Publish/asynchronous response - a service publishes a request to one or recipients, some of whom send back a reply
- Benefits:
  - Loose runtime coupling since it decouples the message sender from the consumer
  - Improved availability since the message broker buffers messages until the consumer is able to process them
  - Supports a variety of communication patterns including request/reply, notifications, request/async response, publish/subscribe, publish/async response etc
- Drawbacks: Additional complexity of message broker, which must be highly available
- Issues: Request/reply-style communication is more complex

Pattern: Remote Procedure Invocation (RPI)

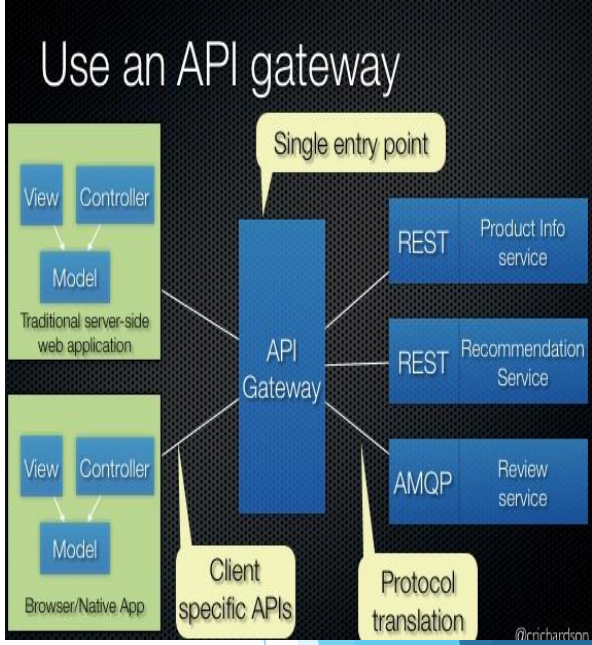
- Use RPI for inter-service communication. The client uses a request/reply-based protocol to make requests to a service.
- Benefits:
  - Simple and familiar
  - Request/reply is easy
  - Simpler system since there in no intermediate broker
- Drawbacks:
  - Usually only supports request/reply and not other interaction patterns such as notifications, request/async response, publish/subscribe, publish/async response
  - Reduced availability since the client and the service must be available for the duration of the interaction
- Issues:
  - Client needs to discover locations of service instances

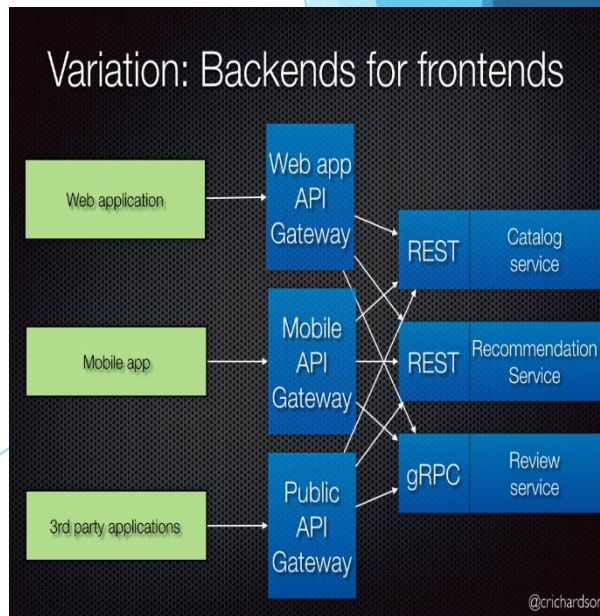
Pattern: Circuit Breaker

- To prevent a network or service failure from cascading to other services, a service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker.
- When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately.
- After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.
- Netflix Hystrix is an example of a library that implements this pattern
- Benefits: Services handle the failure of the services that they invoke
- Issues: It is challenging to choose timeout values without creating false positives or introducing excessive latency.

Pattern: API Gateway / Backends for Frontends

- We implement an API gateway for the single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services.
- Rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client. For example, the Netflix API gateway runs client-specific adapter code that provides each client with an API that's best suited to its requirements.
- The API gateway might also implement security, e.g. verify that the client is authorized to perform the request
- Variation: Backends for frontends
  - A variation of this pattern is the Backends for frontends pattern. It defines a separate API gateway for each kind of client.
- Benefits:
  - Insulates the clients from how the application is partitioned into microservices
  - Insulates the clients from the problem of determining the locations of service instances
  - Provides the optimal API for each client
  - Reduces the number of requests/roundtrips. For example, the API gateway enables clients to retrieve data from multiple services with a single round-trip. Fewer requests also means less overhead and improves the user experience. An API gateway is essential for mobile applications.
  - Simplifies the client by moving logic for calling multiple services from the client to API gateway
  - Translates from a "standard" public web-friendly API protocol to whatever protocols are used internally
- Drawbacks:
  - Increased complexity - the API gateway is yet another moving part that must be developed, deployed and managed
  - Increased response time due to the additional network hop through the API gateway - however, for most applications the cost of an extra roundtrip is insignificant.





Pattern: Single Service Instance per Host

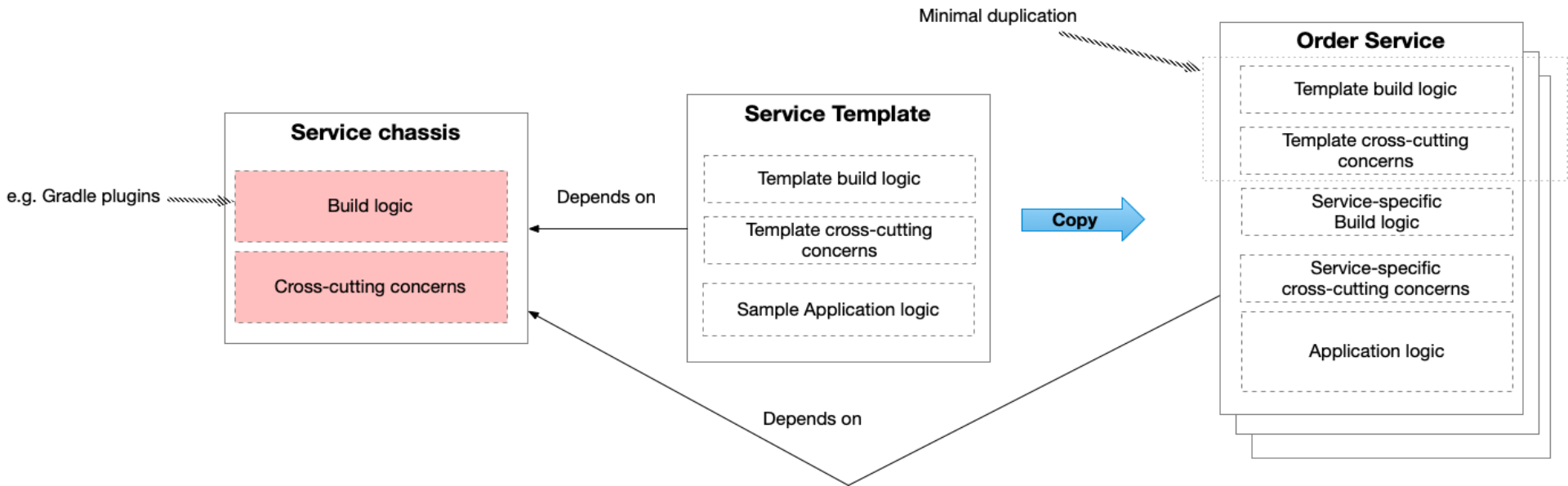
- Here we deploy each single service instance on its own host.
- Benefits:
  - Services instances are isolated from one another
  - There is no possibility of conflicting resource requirements or dependency versions
  - A service instance can only consume at most the resources of a single host
  - Its straightforward to monitor, manage, and redeploy each service instance
- Drawbacks:
  - Potentially less efficient resource utilization compared to Multiple Services per Host because there are more hosts

Pattern: Multiple service instances per host

- Here we run multiple instances of different services on a host (Physical or Virtual machine).
- There are various ways of deploying a service instance on a shared host including:
  - Deploy each service instance as a JVM process. For example, a Tomcat or Jetty instances per service instance.
  - Deploy multiple service instances in the same JVM. For example, as web applications or OSGI bundles.
- Benefits:
  - More efficient resource utilization than the Service Instance per host pattern
- Drawbacks:
  - Risk of conflicting resource requirements
  - Risk of conflicting dependency versions
  - Difficult to limit the resources consumed by a service instance
  - If multiple services instances are deployed in the same process then its difficult to monitor the resource consumption of each service instance. Its also impossible to isolate each instance

Pattern: Microservice chassis

- The chassis implements
  - Reusable build logic that builds, and tests a service. This includes, for example, Gradle Plugins.
  - Mechanisms that handle cross-cutting concerns. The chassis typically assembles and configures a collection of frameworks and libraries that implement this functionality.
- Benefit: The main benefit of a microservice chassis is that it's faster and easier to keep the dependencies, build logic and cross-cutting concern logic up to date. You simply release a new version of the microservice chassis framework that contains the needed changes, and update each service to the use new version.
- Issue: One issue is that you need a microservice chassis for each programming language/framework that you want to use. This can be an obstacle to adopting a new programming language or framework.



End of Topic

SB Microservices Patterns