Part I- Foundation

- Chapter 1 : Introduction to Microservices
  - Definition:
    - Microservices are independently releasable services that are modeled around a business domain.
    - A service encapsulates functionality and makes it accessible to other services via networks—you construct a more complex system from these building blocks
  - Benefits of microservices architecture:
    - It's so Cool 😎
    - Each service can be scaled independently to meet demand
    - Different services can use different technologies and programming languages
    - The failure of one service does not necessarily impact others
    - Smaller, autonomous teams can develop, test, and deploy services independently
    - Smaller codebases are easier to understand, maintain, and refactor. Each microservice is focused on a specific business capability, which simplifies the code and makes it more manageable.
    - Technology Heterogeneity

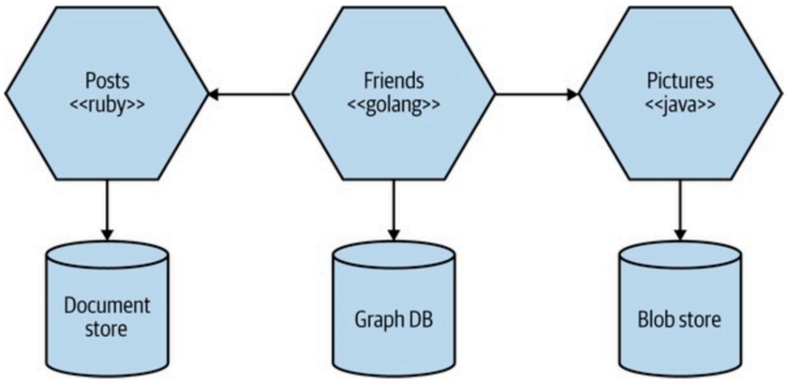

Figure 1-10. Microservices can allow you to more easily embrace different technologies

  - Key concepts of microservices
    - Owning Their Own State:
      - microservices should avoid the use of shared databases
      - Hiding internal state in a microservice

        If a microservice wants to access data held by another microservice, it should go and ask that second microservice for the data
    - Size:
      - James Lewis, technical director at Thoughtworks, has been known to say that “a microservice should be as big as my head.”
      - a microservice should be kept to the size at which it can be easily understood
  - The Monolith: When all functionality in a system must be deployed together, I consider it a monolith
    - The Single-Process Monolith:
      - a system in which all of the code is deployed as a single process:
      - You may have multiple instances of this process for robustness or scaling reasons, but fundamentally all the code is packed into a single process
  - Microservice Pain Points
    - Developer Experience: As you have more and more services, the developer experience can begin to suffer. More resource-intensive runtimes like the JVM can limit the number of microservices that can be run on a single developer machine
    - Technology Overload

    - **Cost**
        - Firstly, you'll likely need to run more things—more processes, more computers, more network, more storage, and more supporting software (which will incur additional license fees).
        - Secondly, any change you introduce into a team or an organization will slow you down in the short term. It takes time to learn new ideas, and to work out how to use them effectively
        - In my experience, microservices are a poor choice for an organization primarily concerned with reducing costs, as a cost-cutting mentality—where IT is seen as a cost center rather than a profit center—will constantly be a drag on getting the most out of this architecture. On the other hand, microservices can help you make more money if you can use these architectures to reach more customers or develop more functionality in parallel. So are microservices a way to drive more profits? Perhaps. Are microservices a way to reduce costs? Not so much.
    - Reporting: hard to join multiple different databases physically. With a monolithic system, you typically have a monolithic database. This means that stakeholders who want to analyze all the data together, often involving large join operations across data, have a ready-made schema against which to run their reports.

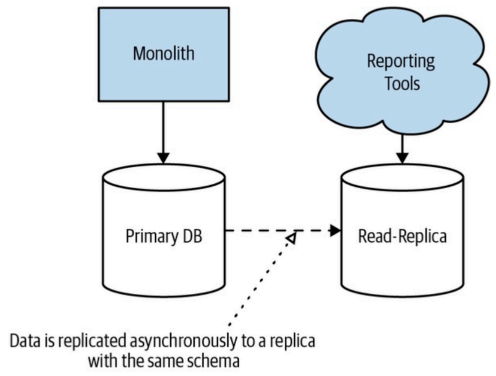
Monolith
Reporting Tools
Primary DB
Read-Replica
Data is replicated asynchronously to a replica with the same schema

Figure 1-12. Reporting carried out directly on the database of a monolith

    - Monitoring and Troubleshooting:
        - With a standard monolithic application, we can have a fairly simplistic approach to monitoring. We have a small number of machines to worry about, and the failure mode of the application is somewhat binary—the application is often either all up or all down
        - With a microservice architecture, do we understand the impact if just a single instance of a service goes down
        - With a monolithic system, if our CPU is stuck at 100% for a long time, we know it's a big problem. With a microservice architecture with tens or hundreds of processes, can we say the same thing? Do we need to wake someone up at 3 a.m. when just one process is stuck at 100% CPU?
    - Security
    - Testing
    - Latency: With a microservice architecture, processing that might previously have been done locally on one processor can now end up being split across multiple separate microservices. Information that previously flowed within only a single process now needs to be serialized, transmitted, and deserialized over networks
    - Data Consistency: Shifting from a monolithic system, in which data is stored and managed in a single database, to a much more distributed system, in which multiple processes manage state in different databases, causes potential challenges with respect to consistency of data
- Should I Use Microservices?:
    - microservice architectures are often a bad choice for brand-new products or

      startups
        - the domain that you are working with is typically undergoing significant change as you iterate on the fundamentals of what you are trying to build
            - Uber initially focused on limos
            - Flickr spun out of attempts to create a multiplayer online game
        - The process of finding product market fit means that you might end up with a very different product at the end than the one you thought you'd build when you started
        - Startups also typically have fewer people available to build the system, which creates more challenges with respect to microservices. Microservices bring with them sources of new work and complexity, and this can tie up valuable bandwidth

        - For a small team, a microservice architecture can be difficult to justify because there is work required just to handle the deployment and management of the microservices themselves. Some people have described this as the "microservice tax." When that investment benefits lots of people, it's easier to justify. But if one person out of your five- person team is spending their time on these issues, that's a lot of valuable time not being spent building your product. It's much easier to move to microservices later
        - creating software that will be deployed and managed by their customers may struggle with microservices
    - microservice architectures work well
        - probably the single biggest reason that organizations adopt microservices is to allow for more developers to work on the same system without getting in each other's way
        - A five-person startup is likely to find a microservice architecture a drag.
        - A hundred-person scale-up that is growing rapidly is likely to find that its growth is much easier to accommodate with a microservice architecture properly aligned around its product development efforts.
        - Software as a Service (SaaS) applications are, in general, also a good fit for a microservice architecture
            - These products are typically expected to operate 24-7, which creates challenges when it comes to rolling out changes
            - Furthermore, the microservices can be scaled up or down as required

- Chapter2: Modeling Microservices
    - MusicCorp:
        - Business Domain: MusicCorp is an online music store that sells digital music tracks and albums. It provides services such as browsing music, purchasing tracks, and managing user accounts
        - Microservices Implementation: The company transitions from a monolithic application to a microservices architecture. Each microservice represents a specific business capability, such as the music catalog, user accounts, and the purchasing system
    - What Makes a Good Microservice Boundary
        - Information Hiding: Each microservice should encapsulate its internal implementation details and expose only what is necessary through a well-defined interface. This reduces dependencies and allows services to evolve independently.
        - Cohesion: A microservice should have a high degree of cohesion. making sure that everything a microservice does is part of the same job or function.
        - Coupling: Services should be loosely coupled, meaning changes in one service should not significantly impact others
    - Types of Coupling: Coupling refers to the degree of direct knowledge that one module has about another

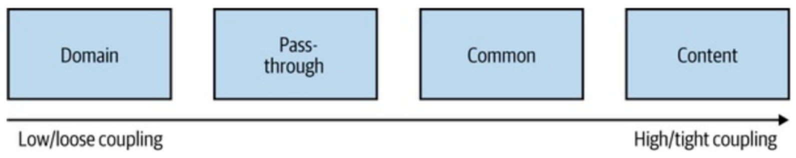

Figure 2-1. The different types of coupling, from loose (low) to tight (high)

        - Domain Coupling(OK): Occurs when one microservice needs to interact with another to use its functionality. MusicCorp's Order Processor calls Warehouse and Payment services.Although necessary, excessive domain coupling can indicate that a service is doing too much.
        - Temporal Coupling: When one service requires another service to be available and responsive at the same time for an operation to complete. Order Processor making a synchronous HTTP call to Warehouse, Can lead to blocking and resource contention. Asynchronous communication can mitigate this.
        - Common Coupling(BAD): Multiple microservices share the same data source (e.g., a shared database). Order Processor and Warehouse both accessing and updating the same order table. in shared data structure can affect all services. Common coupling can lead to resource contention and operational difficulties
        - Content Coupling(BAD): An upstream service directly accesses and modifies the internal state of a downstream service. Warehouse service directly updates the Order service's database. This is highly undesirable as it exposes internal details, leading to a loss of modularity and making changes difficult without widespread impact
    - Just Enough Domain-Driven Design:

    - Why Use Domain-Driven Design (DDD) for Microservices
        - Alignment with Business Needs: DDD ensures that the software closely aligns with business requirements. By focusing on the core business domain and its complexities, DDD helps in creating microservices that directly address business needs.
        - Clear Service Boundaries: DDD introduces the concept of bounded contexts, which define clear boundaries around different parts of the system. These boundaries help in determining where one microservice ends and another begins, ensuring that each service has a well-defined scope and responsibility.
        - Improved Communication: The use of ubiquitous language in DDD ensures that both developers and business stakeholders use the same terminology. This common language improves communication and reduces misunderstandings, leading to a more accurate representation of business processes in the code.
    - Ubiquitous Language: Instead of generic terms like "arrangement" for various banking operations, use specific terms like "loan," "share," or "credit card."
- Chapter 3: Splitting the Monolith: Many of you will already have an existing system, perhaps some form of monolithic architecture, which you are looking to migrate to a microservice architecture.
    - Have a Goal: Microservices are not the goal. You don't "win" by having microservices. Adopting a microservice architecture should be a conscious decision, one based on rational decision making. You should be thinking of migrating to a microservice architecture only if you can't find any easier way to move toward your end goal with your current architecture. I've seen teams obsessed with creating microservices without ever asking why. This is problematic in the extreme given the new sources of complexity that microservices can introduce.
    - Incremental Migration: Avoid big-bang rewrites; instead, break down the monolith gradually
    - The Monolith Is Rarely the Enemy:
        - Monolithic architectures are not inherently bad and can coexist with microservices
        - Focus on the benefits of microservices rather than eliminating the monolith completely
    - The Dangers of Premature Decomposition:
        - Prematurely breaking down a system into microservices without understanding the domain can lead to costly mistakes
        - Start with a monolith to understand boundaries better before transitioning
    - What to Split First
        - Prioritize functionality that helps achieve your goals, such as scaling or improving time to market.
        - Choose areas that are self-contained or cause the most constraints in the monolith
    - Decomposition by Layer
        - Code first

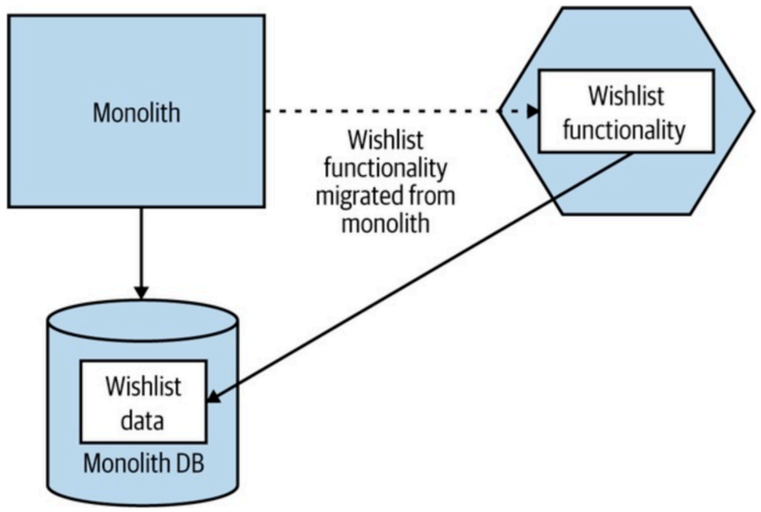

Figure 3-3. Moving the wishlist code into a new microservice first, leaving the data in the monolithic database

        - Data first

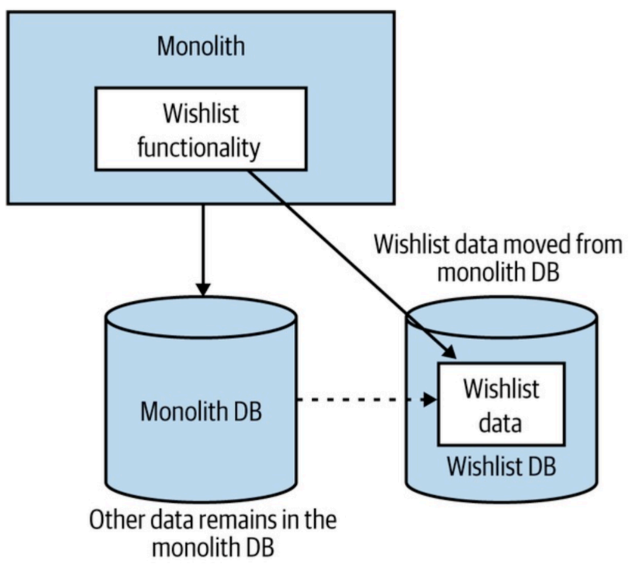
Monolith
Wishlist functionality
Wishlist data moved from monolith DB
Monolith DB
Wishlist data
Wishlist DB
Other data remains in the monolith DB

Figure 3-4. The tables associated with the wishlist functionality are extracted first

- **Useful Decompositional Patterns**
    - Strangler Fig Pattern

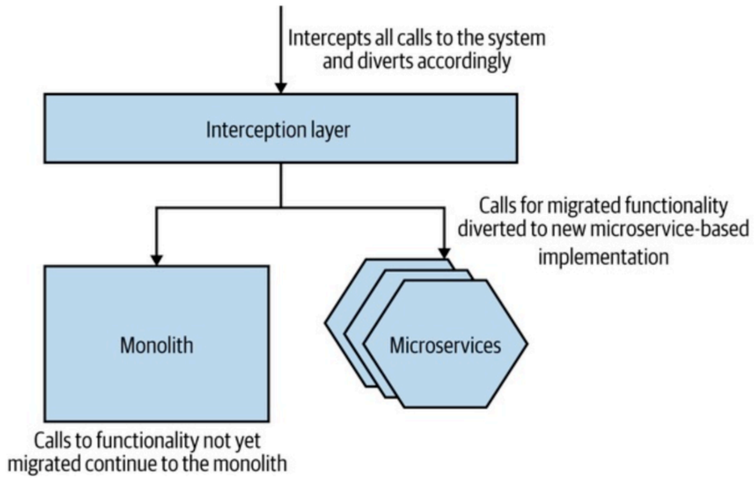
Intercepts all calls to the system and diverts accordingly
Interception layer
Calls for migrated functionality diverted to new microservice-based implementation
Monolith
Microservices
Calls to functionality not yet migrated continue to the monolith

Figure 3-5. An overview of the strangler fig pattern

        - Gradually replace parts of the monolith by wrapping it with new microservices.
    - Parallel Run
        - Run both monolithic and microservice implementations side by side to ensure reliability.
    - Feature Toggle
        - Use toggles to switch between old and new implementations during transition.
- Data Decomposition Concerns:When we start breaking databases apart, we can cause a number of issues. Here are a few of the challenges you might face, and some tips to help
    - Performance: Moving joins from the database to the application layer can decrease performance.
    - Data Integrity: Ensure data integrity across services without relying on database constraints.
    - Transactions: Manage state changes without traditional ACID transactions, possibly using sagas.
    - Tooling: Use tools like Flyway or Liquibase to manage schema changes during migration
- Reporting Database:

        - Create a dedicated reporting database to handle external access and maintain information hiding.
        - Ensure the reporting database schema is designed for consumer needs and is maintained by the microservice team
- Chapter 4: Communication Styles: Understanding the different communication styles between microservices is crucial for designing efficient and reliable systems. This chapter breaks down the various communication methods, their pros and cons, and how to choose the right approach for your needs
    - From In-Process to Inter-Process
        - Performance: Inter-process calls are slower than in-process calls due to network overhead
        - Changing Interfaces: Modifying interfaces is more complex in microservices as changes must be coordinated across independently deployable units
        - Error Handling: Distributed systems face different types of failures, such as crash failures, omission failures, timing failures, response failures, and arbitrary (Byzantine) failures
    - Styles of Microservice Communication:

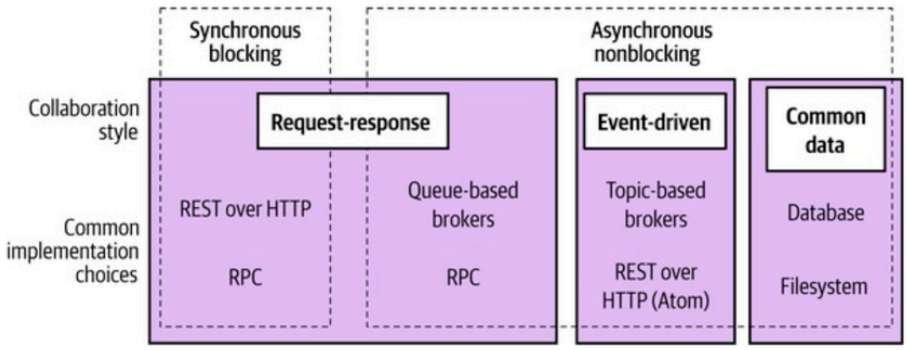

Figure 4-1. Different styles of inter-microservice communication along with example implementing technologies

        - Synchronous Blocking: The sender waits for a response
        - Asynchronous Nonblocking: The sender continues processing without waiting
        - Request-Response: The sender expects a reply after requesting an action
        - Event-Driven: Microservices react to events without direct interaction
        - Common Data: Microservices collaborate through shared data sources
    - Pattern: Synchronous Blocking:

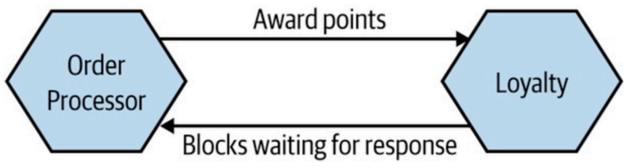

Figure 4-2. `Order Processor` *sends a synchronous call to the* `Loyalty` *microservice, blocks, and waits for a response*

        - Advantages:
            - Familiar and straightforward
            - Simplifies sequential operations
        - Disadvantages:
            - Temporal coupling: Both services must be available simultaneously
            - Can lead to cascading failures and resource contention
        - Usage
            - Suitable for simple architectures and short call chains
            - Problematic with long chains of dependencies

- Pattern: Asynchronous Nonblocking:

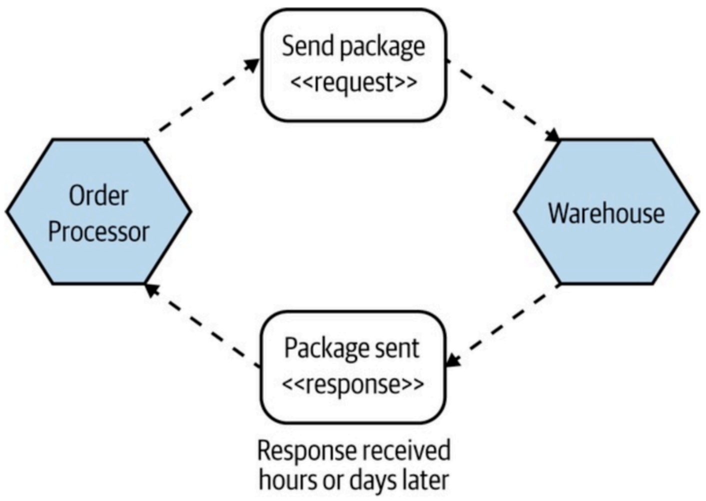

Figure 4-5. The `Order Processor` kicks off the process to package and ship an order, which is done in an asynchronous fashion

  - Advantages:
    - Decouples services temporally, reducing dependencies
    - Handles long-running processes efficiently
  - Disadvantages:
    - Increased complexity and varied implementation styles: Message Brokers and Queues, Event-Driven Architecture, Pub/Sub, Callback Mechanisms, Polling,…
    - Potential challenges with message correlation and state management
      - Message correlation refers to the ability to link a response message back to the original request message in a system where messages are sent and received asynchronously. This is crucial for ensuring that the correct response is matched with the correct request, especially when multiple messages are in transit simultaneously
      - When a service sends an asynchronous request, it often expects a response. The system must have a way to identify and correlate this response with the original request
      - This typically involves generating a unique identifier (correlation ID) for each request, which is then included in the response
  - Usage: Ideal for long-running processes and complex workflows
- Pattern: Communication Through Common Data

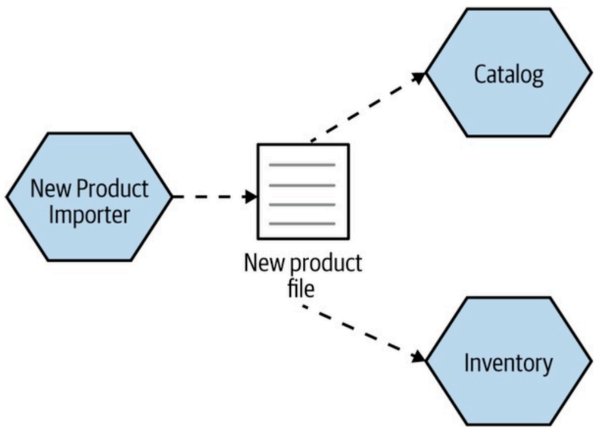

Figure 4-6. One microservice writes out a file that other microservices make use of

  - Advantages
    - Simple implementation using familiar technology

        - Disadvantages
            - Polling mechanisms can introduce latency
            - Risk of coupling if data structures change
        - Usage: Effective for high-volume data sharing and legacy system integration
    - Pattern: Request-Response Communication
        - Advantages
            - Ensures that operations complete before proceeding
            - Useful for actions requiring confirmation or data retrieval
        - Disadvantages
            - Requires careful handling of timeouts and retries
            - Can lead to blocking issues if implemented synchronously
        - Usage: Suitable for operations needing immediate feedback or confirmation
    - Pattern: Event-Driven Communication

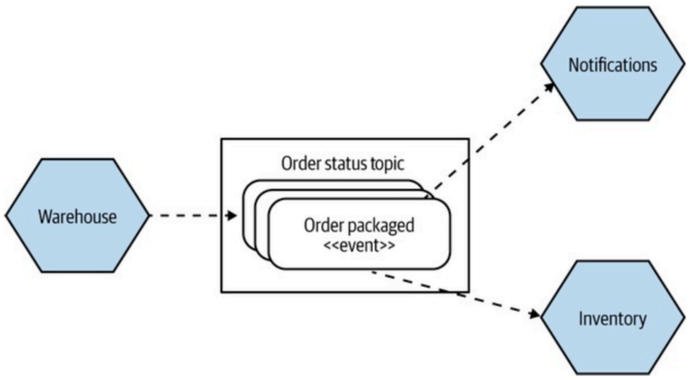

Figure 4-11. The `Warehouse` emits events that some downstream microservices subscribe to

        - Advantages
            - Loose coupling between services
            - Enables asynchronous processing and scalability
        - Disadvantages
            - Complexity in managing event propagation and handling
            - Potential issues with event size and data sensitivity
        - Usage: Best for broadcasting information and handling complex workflows