Chapter 21

Microservices

1. Introduction: From Monolith to Microservices

- As monolithic applications grow to satisfy more business requirements, components are continually added.

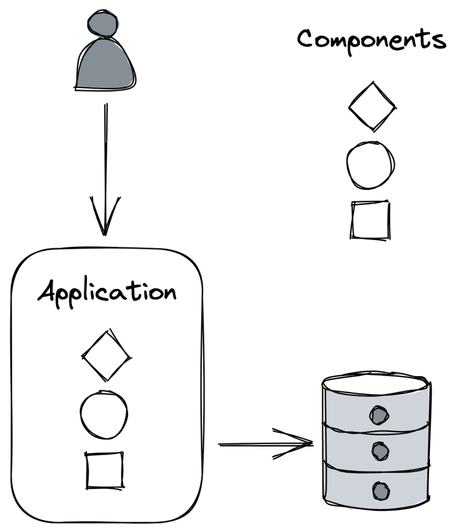

Figure 1: A monolithic application composed of multiple components

- **Problems with Growing Monolithic Applications:**
  - Components often become *increasingly coupled* over time, leading to developers stepping on each other's toes more frequently.
  - The codebase can become so complex that *nobody fully understands every part*, making new feature implementation and bug fixing very time-consuming.
  - A change to a single component might necessitate the *entire application to be rebuilt and redeployed.*
  - If a new deployment introduces a bug (e.g., a memory or socket leak), *unrelated components can be affected.*

  – Reverting a problematic deployment *impacts the velocity of all developers*, not just the one who introduced the bug.

- **Solution: Microservice Architecture**
  – Functionally decompose the monolithic application into a set of *independently deployable services* that communicate via APIs.

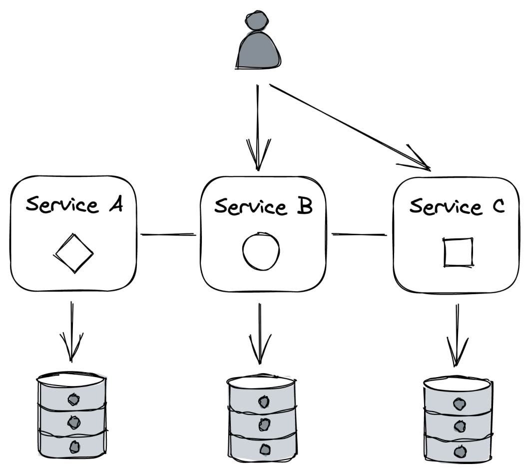

Figure 2: An application split into independently deployable services that communicate via APIs

- **APIs as Boundaries:** APIs decouple services from each other by creating *hard boundaries* that are more difficult to violate than those between components running in the same process.

- **Organizational & Technical Benefits:**

  – **Team Structure:** Each service can be *owned and operated by a small team.* Smaller teams collaborate more effectively due to reduced communication overhead (which grows quadratically with team size).
  – **Autonomy & Reduced Cross-Team Communication:** Each team controls its own codebase and dictates its own release schedule, requiring less overall cross-team communication.
  – **Manageable Scope:** The surface area of a single service is smaller than the whole application, making it *more digestible* for developers,

especially new hires.

  - **Technological Freedom:** In principle, each team is free to adopt the *tech stack and hardware* that best fits their service's specific needs, as API consumers don't care about implementation details.
  - **Experimentation:** Makes it easier to experiment with and evaluate *new technologies* without affecting other parts of the system.
  - **Data Model Independence:** Each service can have its own *independent data model and data store(s)* tailored to its use cases.

- **Terminology - "Micro" in Microservices:**

  - The term "micro" can be misleading; services *don't have to be tiny.*
  - If a service doesn't do much, it only adds operational overhead and complexity.
  - **Rule of Thumb:** APIs should have a *small surface area* but encapsulate a *significant amount of functionality.*

2. Caveats of Microservices ("Microservice Premium")

- Splitting an application into services adds *a great deal of complexity* to the overall system. This "premium" is only worth paying if the benefits can be amortized across many development teams.
- **Tech Stack Proliferation vs. Standardization:**
  - While freedom in tech stack choice is a benefit, it can make it *more difficult for developers to move* between teams.
  - It also leads to a *sheer number of libraries* (one for each adopted language) that need to be supported for common functionalities like logging.
  - **Solution:** Enforce a certain degree of standardization, often by *loosely*

encouraging specific technologies through providing excellent development experience and support for a recommended portfolio.

- **Communication Overhead & Complexity:**
  - *Remote calls between services are expensive* and introduce non-determinism (network issues, latency).
  - While a monolith also deals with external communication (client requests, third-party APIs), these issues are *amplified* in a microservice architecture due to the increased number of internal service-to-service calls.
- **Coupling (Risk of Distributed Monolith):**
  - Microservices *must be loosely coupled* so that a change in one service doesn't necessitate changes in others.
  - If services are tightly coupled, you can end up with a **distributed monolith**, which has all the downsides of a monolith plus the complexity of a distributed system.
  - **Causes of Tight Coupling:** Fragile APIs requiring clients to update on any change, shared libraries that must be updated in lockstep across services, or using static IP addresses for service references.
- **Resource Provisioning:**
  - Supporting many independent services requires a *simple and efficient way* to provision new machines, data stores, and other commodity resources.
  - You don't want every team inventing its own provisioning methods.
  - A fair amount of *automation* is needed for efficient configuration and management.
- **Testing:**
  - Testing *individual* microservices is not necessarily more challenging than

testing components of a monolith.
  - However, *testing the integration of microservices is a lot harder.* Subtle and unexpected behaviors often emerge only when services interact with each other at scale in a production-like environment.

- **Operations:**
  - A *common way of continuously delivering and deploying* new builds safely to production is needed, so each team doesn't have to reinvent the wheel.
  - *Debugging failures, performance degradations, and bugs is significantly more challenging* with microservices because you can't just load the whole application onto a local machine and step through it with a debugger.
  - A good **observability platform** (for logging, tracing, metrics) becomes crucial.

- **Eventual Consistency:**
  - Splitting an application into services means the overall data model no longer resides in a single data store; it's spread out.
  - Atomically updating data across different data stores while guaranteeing strong consistency is *slow, expensive, and hard to get right.*
  - Consequently, microservice architectures usually require embracing **eventual consistency** for data spread across services.

- **General Recommendation:**
  - It's often best to *start with a monolith* and ensure it's well-componentized.
  - Decompose the monolith into microservices *only when there is a good reason to do so* (e.g., when organizational scaling issues or deployment complexities become significant).

- This approach allows easier movement of boundaries as the application grows initially. Once the monolith is mature and growing pains arise, you can start to *peel off one microservice at a time*.

3. API Gateway

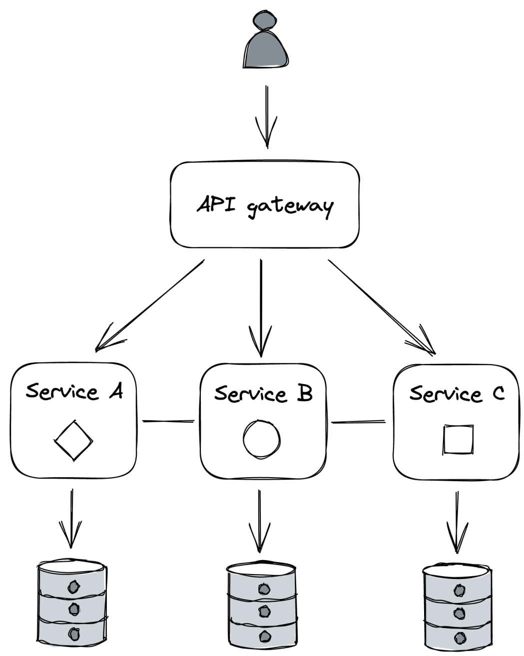

Figure 3: The API gateway hides the internal APIs from its clients

- After decomposing an application into services, how the outside world communicates with it needs rethinking.
- **Problems with Direct Client-to-Internal-Service Communication:**
  - Clients might need to make *multiple requests to different services* to gather all information for a single operation (inefficient, especially for mobile devices consuming battery life).
  - Clients become *aware of internal implementation details*, such as the DNS names of all internal services.
  - This makes it challenging to *change the application's internal architecture*, as it would require changing clients as well (difficult if you don't

control the clients).

  - Public APIs, once released, often need to be *maintained for a very long time*.

- **Solution: API Gateway**
  - Introduce a layer of indirection. The API gateway acts as a *facade or proxy* for the internal services, exposing a single, public API.
  - It is essentially a specialized *reverse proxy*.

A. Core Responsibilities of an API Gateway

- **Routing:**
  - The most obvious function: routing inbound requests to the appropriate internal services.
  - Often implemented using a *routing map* that defines how the public API endpoints map to internal service APIs.
  - This mapping allows internal APIs to *change without breaking external clients*. If an internal endpoint changes, the public endpoint can remain the same, with only the gateway's mapping needing an update.
- **Composition:**
  - In a distributed system, data is spread across multiple services, each with its own data store.
  - Some use cases may require *stitching data together* from these multiple sources.
  - The API gateway can offer a *higher-level API* that queries multiple internal services and *composes their responses* into a single response for the client.
  - **Benefits:** Relieves the client from knowing which services to query and

reduces the number of network requests the client needs to make.

  - **Challenges:**
    * The *availability of the composed API decreases* as the number of internal service calls increases (each call has a non-zero probability of failure).
    * Data might be *inconsistent* across services if updates haven't propagated everywhere yet. The gateway might need logic to resolve such discrepancies.

- **Translation:**
  - The API gateway can translate from one *Inter-Process Communication (IPC) mechanism to another* (e.g., translating an external RESTful HTTP request into an internal gRPC call).
  - It can also *expose different APIs tailored to different clients* or use cases.
    * Example: A desktop application API might return more data than a mobile API due to screen estate. Mobile clients might also need requests batched to reduce battery usage.
  - **Graph-based APIs (e.g., GraphQL):**
    * An increasingly popular solution for providing flexible data fetching.
    * The gateway exposes a *schema* (composed of types, fields, and relationships) that describes the available data.
    * Clients send *queries declaring precisely what data they need.*
    * The gateway's job is to translate these queries into the necessary internal API calls and compose the response.
    * **Benefits:** Reduces development time as there's less need to introduce many different specific API endpoints for various use cases. Clients are free to specify their exact data requirements.
    * GraphQL is a popular technology in this space.

B. Cross-Cutting Concerns

- As a reverse proxy, the API gateway is a suitable place to implement cross-cutting functionality that would otherwise need to be part of each individual service.
- **Examples:**
  - Caching frequently accessed resources.
  - Rate-limiting requests to protect internal services from being overwhelmed.
- **Authentication and Authorization:**
  - These are common and critical cross-cutting concerns.
  - **Authentication:** The process of validating that a *principal* (a human or an application issuing a request) is who it claims to be.
  - **Authorization:** The process of granting an *authenticated principal* permissions to perform specific operations (e.g., create, read, update, delete) on a particular resource, often implemented by assigning roles with specific permissions.
  - **Monolithic Approach (Sessions):**
    * HTTP is stateless, so applications need a way to store data between requests to associate them.
    * On first request, the application creates a *session object* with an ID (e.g., a cryptographically strong random number) and stores it (in-memory cache or external data store).
    * The session ID is returned to the client via an *HTTP cookie*, which the client includes in all future requests.
    * The application can retrieve the session object using the cookie.
    * On successful login, the principal's ID and roles are stored in the

session object, which is later used for authorization decisions.

- **Challenges in Microservices:** It's not obvious which service should be responsible for authentication/authorization when request handling spans multiple services.
- **Common Microservice Approach:**
  * **API Gateway for Authentication:** The API gateway authenticates *external requests* as they are the point of entry. This centralizes logic for different authentication mechanisms and hides complexity from internal services.
  * **Individual Services for Authorization:** Authorizing requests is best left to *individual services* to avoid coupling the API gateway with domain-specific logic.
- **Security Tokens:**
  * After API gateway authentication, a *security token* is created and passed with requests to internal services and their dependencies.
  * Internal services validate this token to obtain the principal's identity and roles.
  * **Token Types include:**
    · *Opaque Tokens:* Do not contain information themselves. They require calling an external authentication service to validate and retrieve the principal's information.
    · *Transparent Tokens:* Embed the principal's information within the token itself, eliminating the need for an external validation call but making the revocation of compromised tokens harder. A popular example is the **JSON Web Token (JWT)**, which is a JSON payload containing an expiration date, the principal's identity/roles, and other metadata, signed with a certificate

trusted by internal services so they can validate it without external calls.

  - **API Keys:**
    * Another common authentication mechanism.
    * Custom keys allowing the API gateway to identify the requesting principal and apply limits/permissions.
    * Popular for public APIs (e.g., GitHub, Twitter).

C. Caveats of API Gateway

- **Development Bottleneck:** Can become a bottleneck as it's tightly coupled with the APIs of the internal services it shields. Whenever an internal API changes, the gateway often needs to be modified as well.
- **Operational Overhead:** It's *one more service* that needs to be deployed, scaled, and maintained.
- **Scalability Requirement:** Must scale to handle the aggregate request rate for *all services* behind it.
- **Worthwhile Investment:** Despite caveats, if an application has many services and APIs, the pros of using an API gateway (like providing a unified public interface and handling cross-cutting concerns) generally outweigh the cons.
- **Implementation Options:**
  - *Roll your own:* Using a reverse proxy like NGINX as a starting point.
  - *Managed Solutions:* Use cloud provider offerings like Azure API Management or Amazon API Gateway.