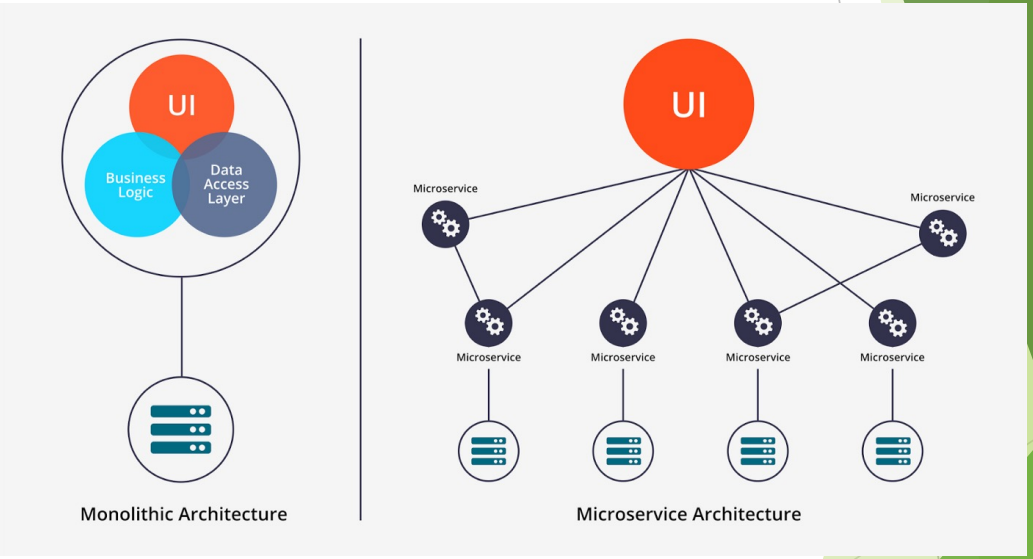
Micro Services

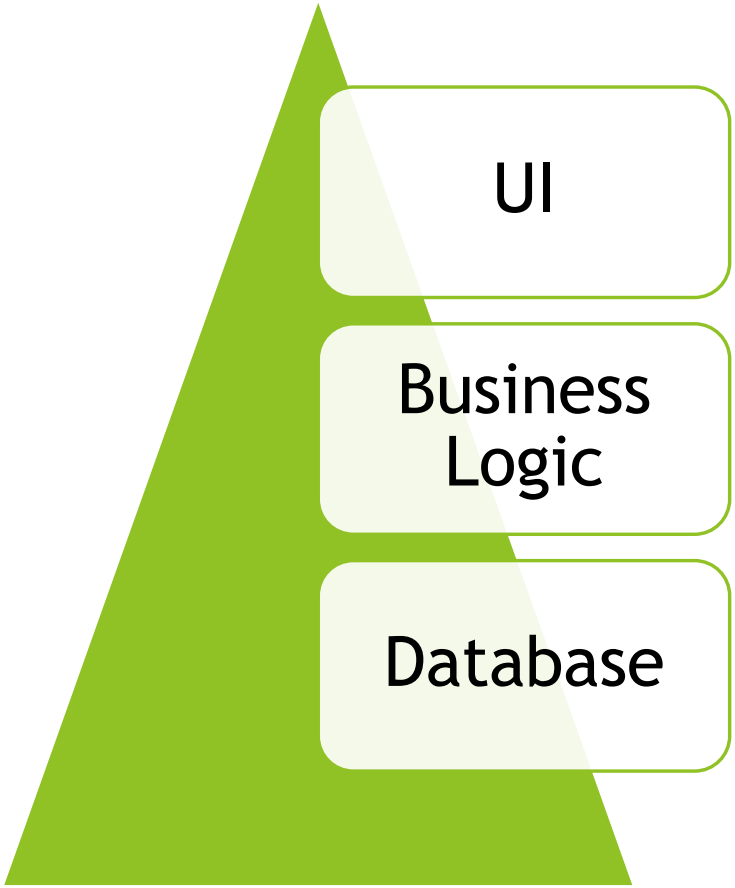
Diogo Gomes dgomes@ua.pt

Introduction

- Microservices architecture has become the most popular architecture in the last decade.
  - It depends on developing small, independent modular services where each service solves a specific problem or performs a unique task and these modules communicate with each other through well-defined API to serve the business goal.



Monolithical Architectures



- Single UI
- Business Logic encompassing various components, interfaces to external services, etc
- Single source of truth (Database)

Monolithic Architecture

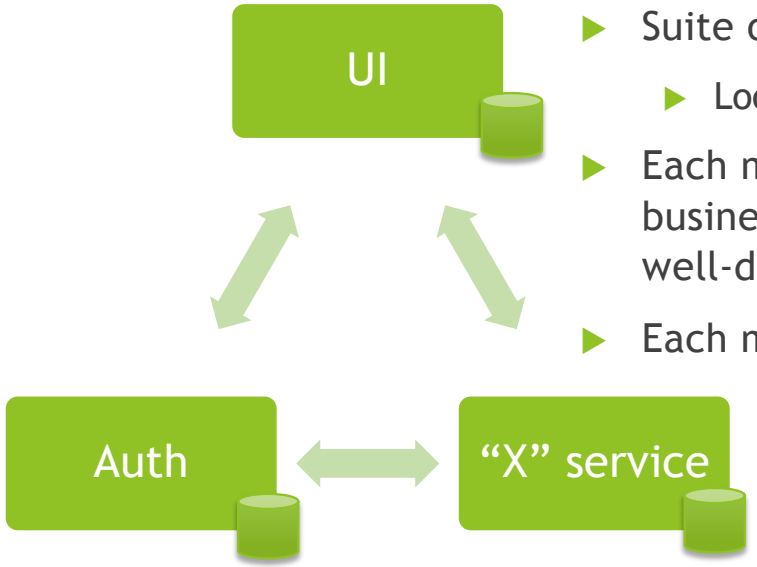
Pros

- Simple to develop
- Simple to test
- Simple to deploy
- Simple to scale horizontally (just add a load balancer)

Cons

- Maintenance
- Size can slow down app
- Redeploy everything on each update
- Hard to scale (specially the database)
- Reliability (any bug brings everything down)
- Hard to adopt to new technologies

Microservices Architectures



- Suite of modules services
  - Loosely coupled
- Each module supports a specific business goal and uses a simple well-defined API
- Each module has its own DB

Microservice Architecture

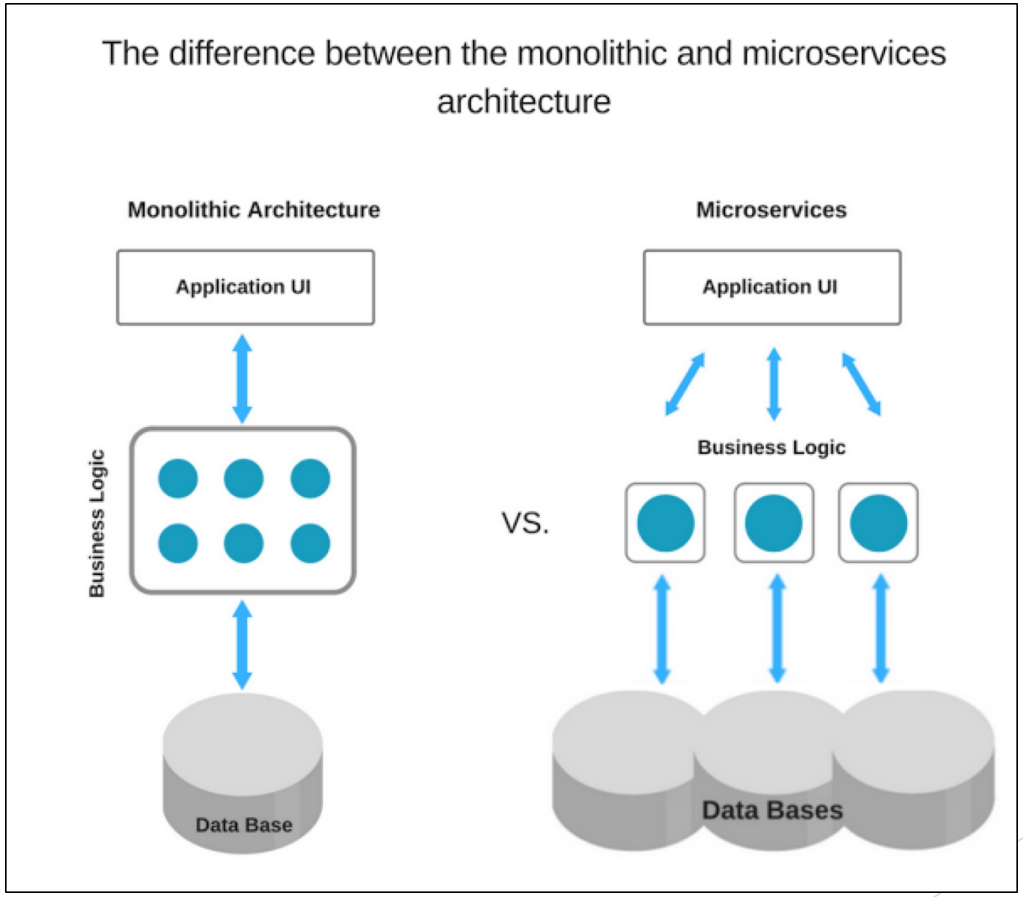
Pros

- Better suited for CI/CD
- Suited for larger distributed teams
- Small -> easier to manage
- Fault Isolation
- Flexible to adopt new technologies

Cons

- Extra complexity of a Distributed System
- Less support from tools (IDE)
- Testing is harder (distributed nature)
- Increased resource usage

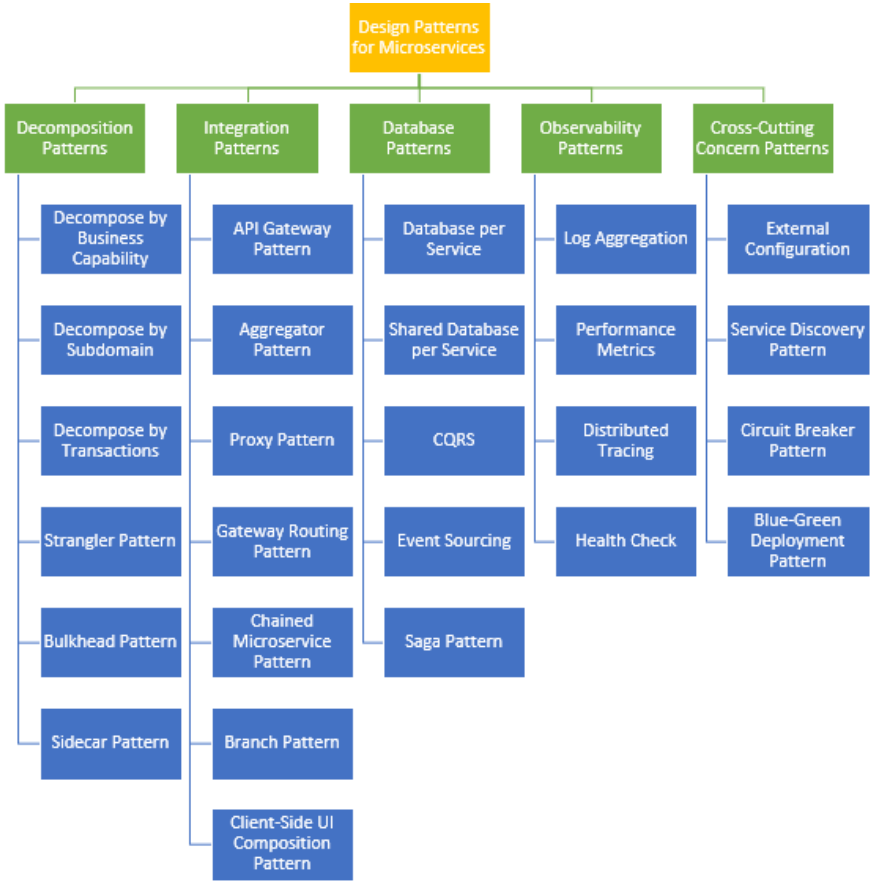
Monolithic vs Microservices



Principles to follow in Microservices Architectures

- Scalability
- Availability
- Resiliency
- Flexibility
- Independent, autonomous
- Decentralized governance
- Failure isolation
- Auto-Provisioning
- Continuous delivery through DevOps

Design Patterns



Decomposition Patterns

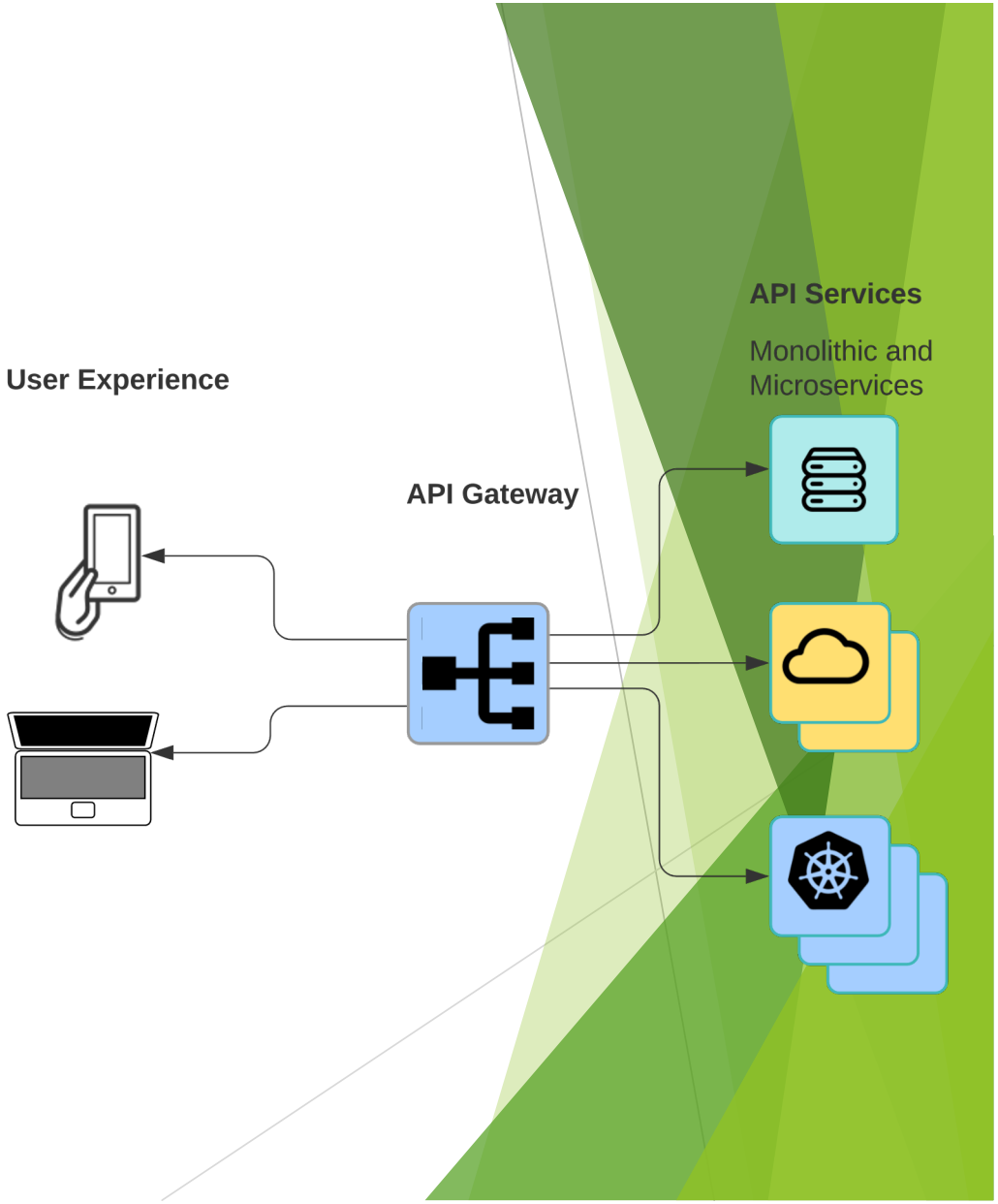
- Decompose by Business Capability
  - E.g. Orders, Costumers, etc
- Decompose by Subdomain
  - E.g. Core, Supporting, etc
- Decompose by Transaction / 2PC pattern
  - E.g. Each microservice handles a transaction (prepare + commit/rollback)
- Strangler Pattern
  - Substitutes existing monolith
  - Transform, coexist, eliminate
- Bulkhead Pattern
  - Isolate as much as possible
- Sidecar Pattern
  - Services are decomposed but kept attached to main process

Integration Patterns

- API Gateway Pattern
  - Single point of entry/routing
  - Transformation/Adaptation
  - Border security
- Aggregator Pattern
  - Aggregate information from diferente components
- Proxy Pattern
  - Similar to API Gateway, but lighter
- Gateway Routing
  - Similar to API Gateway, but lighter
- Chained Microservice
  - Service orchestration
- Branch Pattern
  - Mix of aggregator and chain
- Client-Side UI Composition Pattern

API Gateway Pattern

- 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.



Why use an API Gateway?

- Therefore, having an intermediate level or tier of indirection (Gateway) can be 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 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 approach 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 and SSL. In many situations, those concerns could be handled in a single tier so the internal microservices are simplified.

Main features in the API Gateway pattern

- **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 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.

Main features in the API Gateway pattern

- **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.
- Depending on the API Gateway product you use, it might be able to perform this aggregation. However, in many cases it's more flexible to create aggregation microservices under the scope of the API Gateway, so you define the aggregation in code (that is, C# code):

Main features in the API Gateway pattern

- **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 approach 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 allowlisting

Amazon API Gateway

Database Patterns

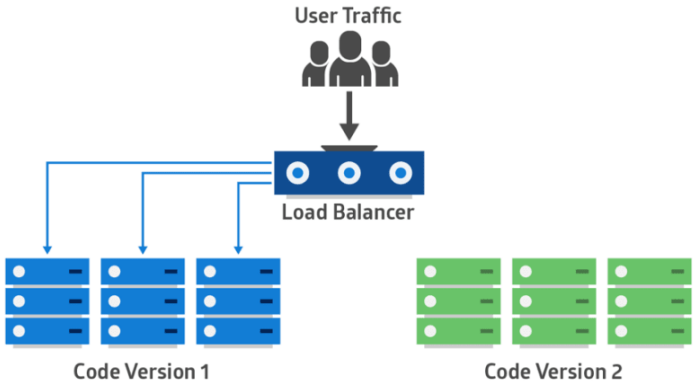
- Database per Service
- Shared Database per Service
  - Brownfield solution
- Command Query Responsibility Segregation (CQRS)
  - Separate CRUD from Query
- Event Sourcing
  - Operations on data are driven by sequences of events
- Saga Pattern
  - Trigger based

Observability Patterns

- Log Aggregation
- Performance Metrics
- Distributed Tracing
- Health Check

Cross-Cutting Concern Patterns

- External Configuration
- Service Discovery Pattern
- Circuit Breaker Pattern
- Blue-Green Deployment Pattern



Referências / Textos de apoio

- https://martinfowler.com/articles/microservices.html
- https://martinfowler.com/microservices/