Architecture & design

Principles and patterns

Jana Kostičová, 28.10.2026

General principles of software architecture

- Separation of Concerns (SoC)
  - Keeping different aspects of the system's functionality or behavior separate and well-defined
  - Each part of the system focuses on specific concerns, making the code more maintainable and understandable
- Modularity
  - Organizing software into discrete, interchangeable components or modules

Component - based architecture
=> reusability, interoperability, encapsulation, maintainability, scalability, …

- Use consistent principles within the components / layers / subsystems
- …

“Modern” principles of software architecture

Avoid Big Design Up Front (BDUF)

- BDUF: Architecture and detailed design are completed before implementation begins
- Often associated with the waterfall model
- Discouraged in modern agile approaches

Build to change instead of build to last

- Avoid over-engineering to accommodate all hypothetical future requirements
- Prefer simple solutions that can be extended or refactored when necessary
- Some decisions are inherently difficult to change later (e.g. architectural style - microservices vs monolith), these should be made with a long-term perspective

What mostly works:

1. System architecture: Primarily up front
    - What are risks of modeling architecture incrementally?
2. Component design: Incrementally

Evaluate strategies

- Model for every possible future need upfront
- Model only for current needs
- Model for current needs while keeping foreseeable future changes inexpensive

Architectural styles and patterns

- Reusables solution to common architectural problems
- Similar to design patterns
- Help communication

Various types of styles / patterns, some of them can be mutually combined

- Deployment
- Structure
- Communication
- Domain
- Network
- ….

Common architectural patterns

Communication patterns

- Client-server
- Peer-to-peer

Application-level patterns

- Service-oriented architecture
- Service-oriented architecture with Enterprise Service Bus
- Microservices architecture

Other

- Layered architecture
- Domain-driven development
- Model-view-controller

Communication patterns in distributed systems

Distributed system: a system whose components run across multiple networked computers.
Common communication patterns:

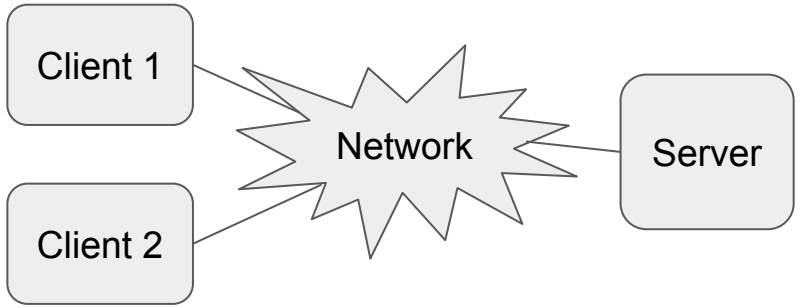
1. Client-server pattern
2. Peer-to-peer pattern
3. Hybrid architecture (combination)

- Define roles and communication relationships between distributed processes
- Primarily reflected in the process view of software architecture
- Describe logical roles, which do not necessarily correspond to separate physical machines

Client-server

- One or more clients connected to a server over a network or internet connection.
- The server hosts, delivers and manages most of the resources and services to be consumed by the client
- Typically follows a request-response pattern (commonly HTTP/HTTPS + Rest, HTTP/HTTPS + GraphQL, …)

- Example: web applications




```
GET /api/products/42 HTTP/1.1
Host: shop.example.com
Accept: application/json
```

The client requests information about product 42.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 42,
  "name": "Running shoes",
  "price": 89.90
}
```

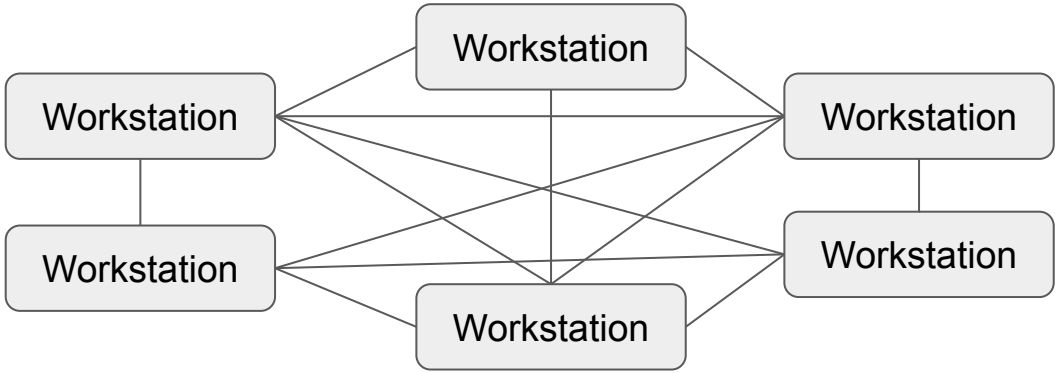
The server returns the product information in JSON format.

Peer-to-peer (P2P)

- Computers, devices, or nodes within a network (peers) communicate and **collaborate directly** with each other without the need for a centralized server or hierarchy of control
- Each peer has equivalent capabilities, and they can act both as clients and servers, sharing resources, data, or services with one another.
- Often central server for e.g., pear discovery, authentication, coordination
- Example: File-sharing system

Communication

- Commonly application-specific protocols (such as BitTorrent). May use TCP or UDP

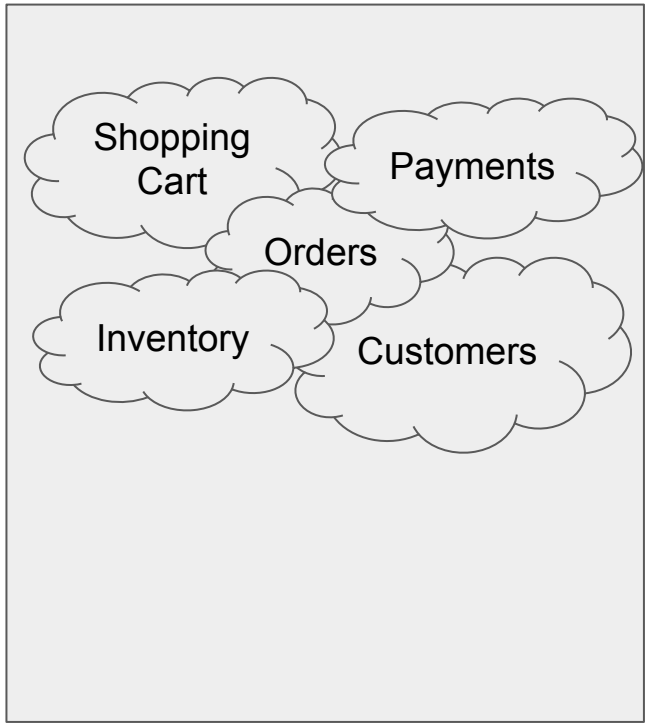


Application-level architectural patterns

1. Monolithic architecture
2. Service-oriented architecture (general)
    a. Service-oriented architecture with Enterprise Service Bus
    b. Microservices architecture

- Covers multiple key aspects of SW architecture - typically reflected in the process, deployment and physical views
- Define how software is decomposed into components, packaged, and deployed, and how those components interact at runtime

Monolithic architecture



- Single unified software application, that is self-contained
  - Single codebase, single runtime process (or a couple of tightly coupled processes), single deployable unit
  - Internal calls (fast)
- Traditional pattern (many legacy systems use it)

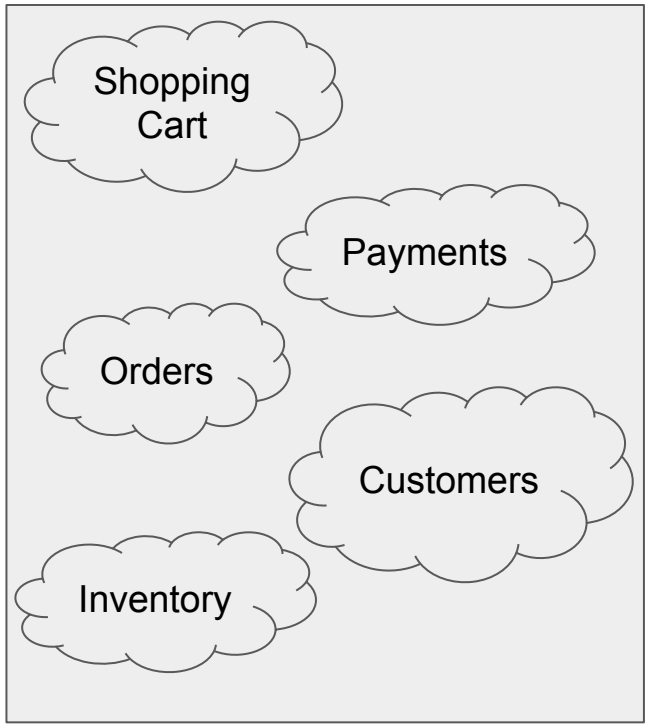
Pros

- Simplicity - everything in single codebase
- Efficiency - fast communication between sub-modules
- Ease of development - running locally, debugging, …

Cons

- Maintainability
- Scalability
- Agility - adding new features can be complex
- Single point of failure

Modular monolith



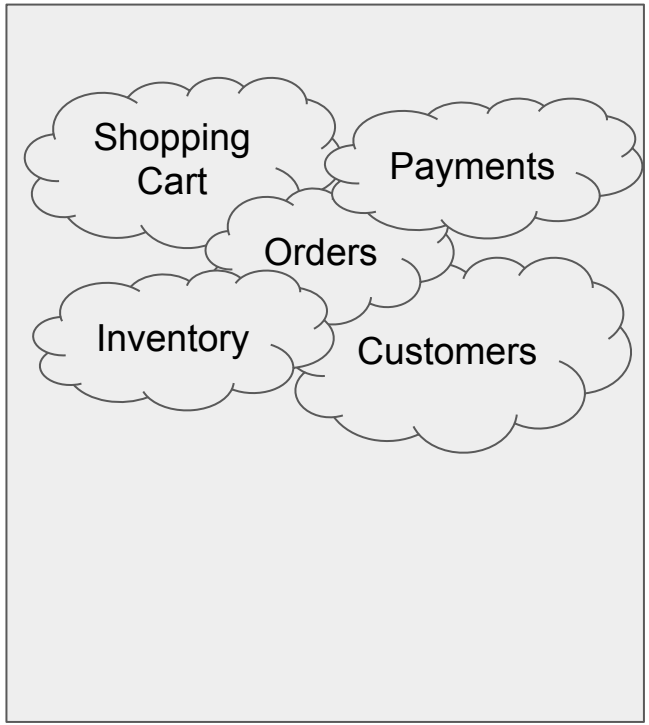
= monolith + good internal modular design

- Has all the defining monolithic characteristics
- In addition: code is organized into independent modules with well defined interfaces

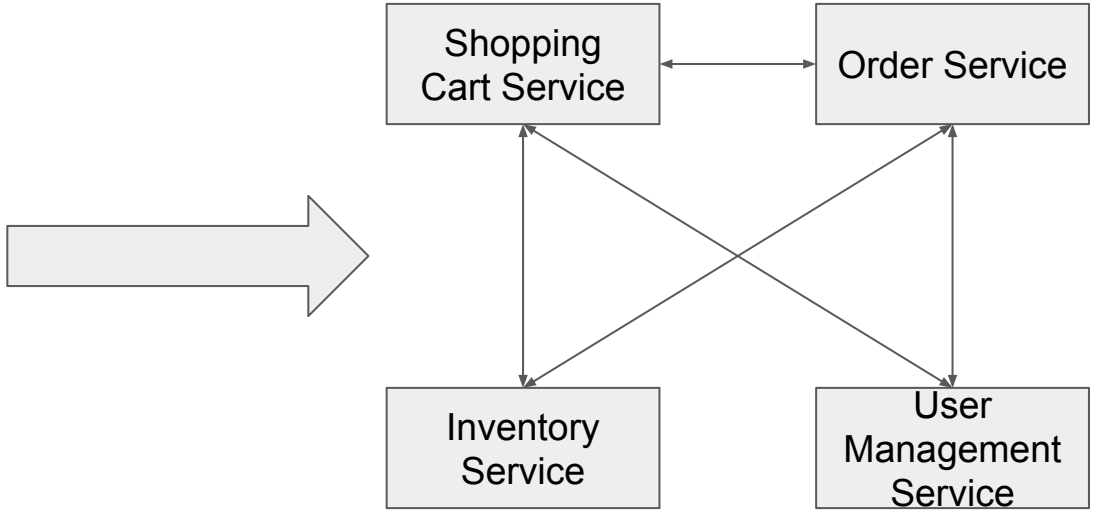
➢ Aligns with component-based architecture

➢ Mitigates some drawbacks of traditional tightly coupled monoliths

Service-Oriented Architecture (SOA)



Monolith



Service-oriented architecture (SOA)

What is a service?

- Many definitions exist

In practice, a service is understood as one of the possible implementations of a software component:

- General characteristics of a software component hold
  - Encapsulates related business functionality
  - Provide well-defined interface
- In addition: a service mostly
  - runs as a separate process
  - makes its API accessible via network
  - is independently deployable

> Definition (Martin Fowler, [16]):
>
> Services are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call.

Often, services are designed to be discoverable and composable.

Monolith vs SOA

Logical components in **monolithic architecture**

- Implementation: Typically as libraries or internal modules
- Communication: Function/method calls within the same process

Logical components in **SOA** (distributed environment)

- Implementation: Exposed as services
- Communication: Over network protocols (REST, SOAP, gRPC)

⇨ SOA = a specialization of component-based architecture where **software components** are **services**

We will talk about two SOA variants: SOA with a bus and microservices.

1970s–1990s
monolithic systems

Late 1990s–2000s
SOA (enterprises)
communication via a bus / SOAP

2005–2010
Early microservices-like systems (Amazon, Netflix,..)

2011–2014
Microservices widely recognized

2017
Microservices in the mainstream

Service-Oriented Architecture - pros and cons

Pros

- Reusability
- Scalability
- Agility
- Loose Coupling
- Platform Independence

Cons

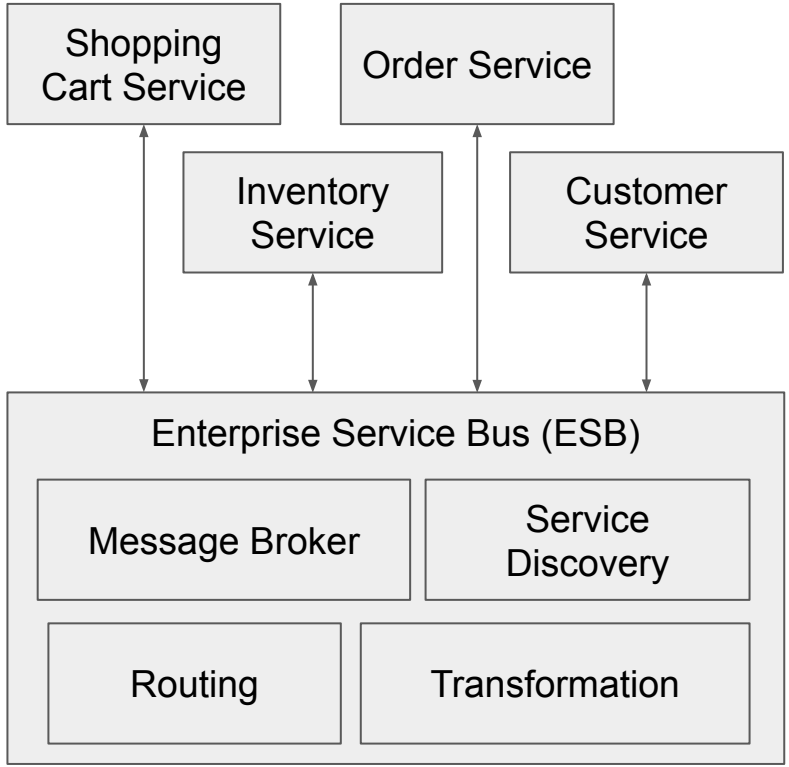
- Complexity (integration, deployment, operation)
- Network overhead (latency, network failures, ..)
- Ease of development - distributed debugging, difficult to run locally
- Security

- Monolith may be sufficient for small and simple systems
- As system grows, it can make sense to refactor to SOA

- SOA with a bus
  pros and cons partially apply
- Microservices
  pros and cons fully apply

Service-oriented architecture with ESB

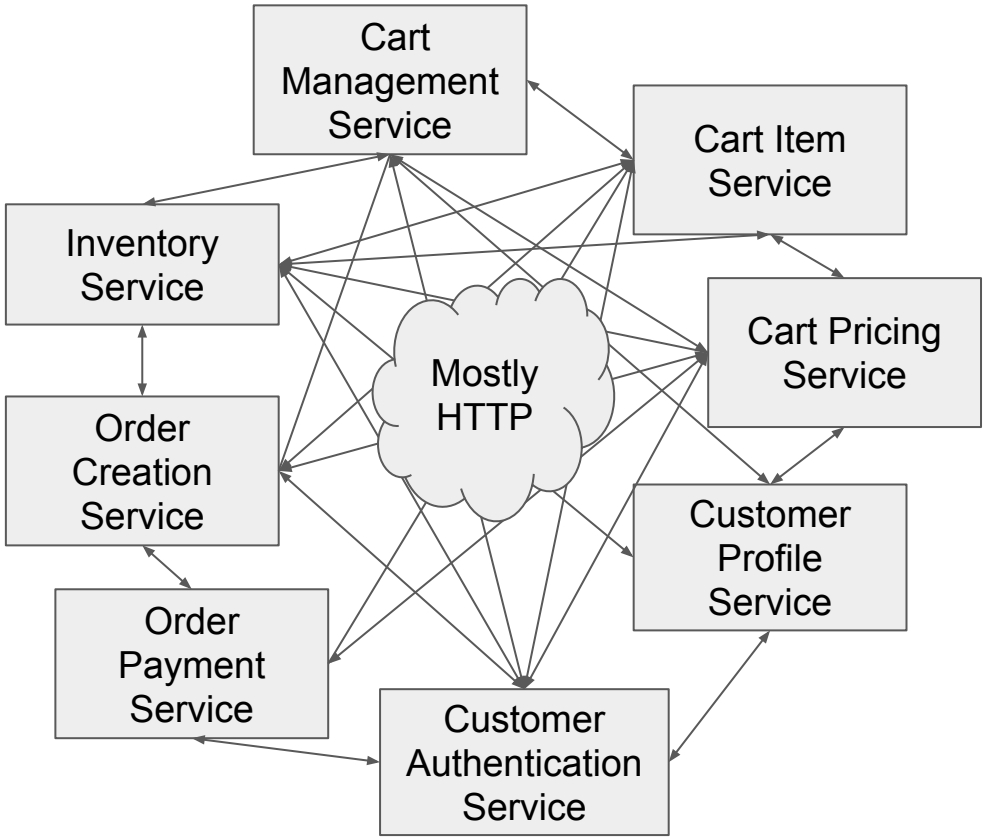
- A variant of SOA
- Services tend to be “large”
  - A single service typically covers a significant business capability
  - Helps to minimize the number of interactions across the bus
- Enterprise Service Bus = central integration point
  - Routing messages between services, transformations, monitoring and control, security, …
  - **Issues:** performance bottleneck, hidden coupling, limited horizontal scalability, only partial flexibility



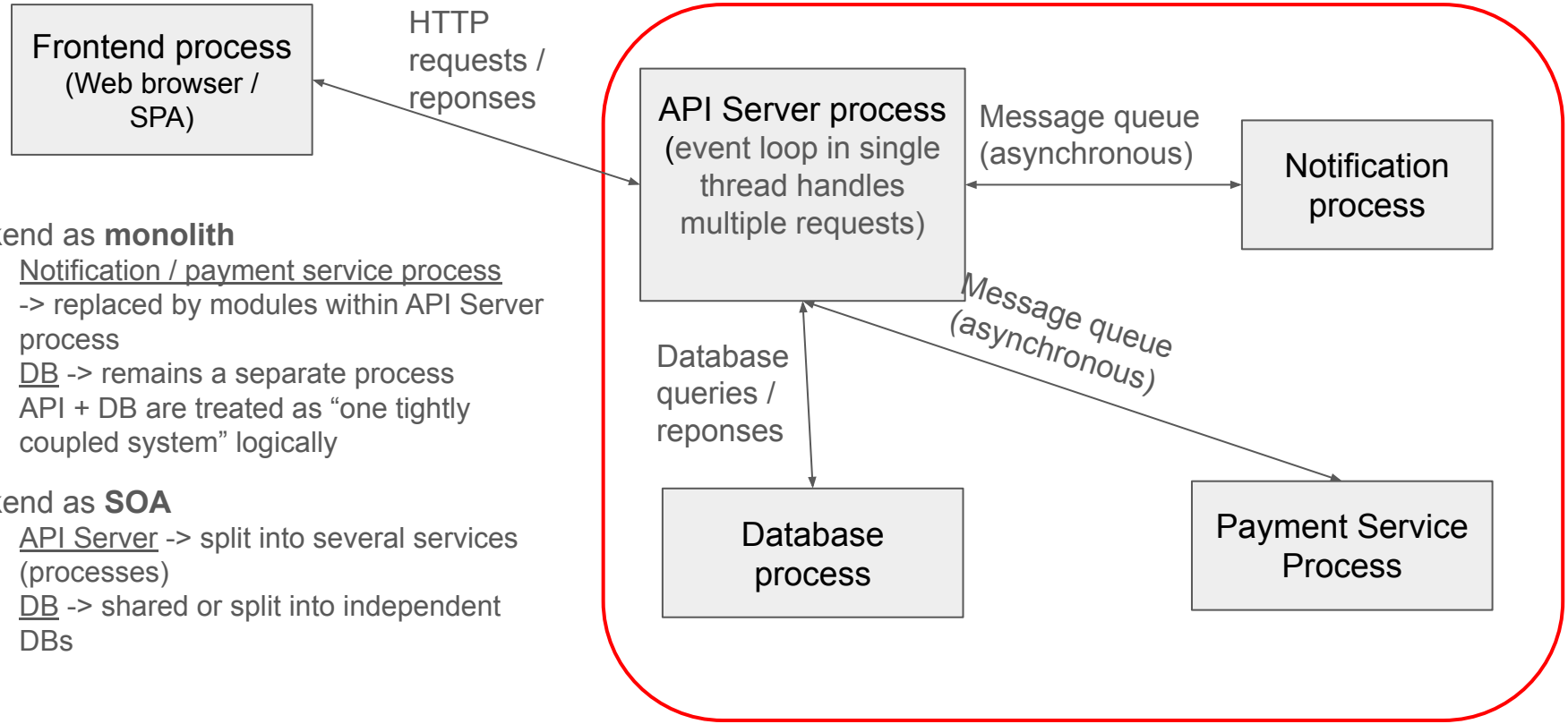
SOA generally may not contain ESB !

Microservice architecture

- A variant of SOA
- Loosely coupled, **fine-grained** (“small”) services - each focuses on one business responsibility
- A microservice is deployed and operates **independently,** it often has its own independent database
- Mutual communication over well-defined APIs
- Commonly used in cloud-native applications
- **Issues:** higher integration, deployment and operational overhead; data consistency



Example: E-commerce store



Backend as **monolith**

- Notification / payment service process
- -> replaced by modules within API Server process
- DB -> remains a separate process
- API + DB are treated as “one tightly coupled system” logically

Backend as **SOA**

- API Server -> split into several services (processes)
- DB -> shared or split into independent DBs

Monolith vs microservices today

Microservices hype has taught us a few lessons:

- Independent services are hard to maintain, test, deploy, and debug
- Operational overhead (orchestration, monitoring, networking) is significant
- Many applications don’t need microservices immediately

Modular monolith - a practical alternative to jumping straight into microservices

Recommended approach (Martin Fowler):

1. Start with a simple design (modular monolith)
2. As complexity grows or requirements change, refactor, extract services, and adapt

→ aligns with moderns principles

- Avoid Big Design Up Front (BDUF)
- Build to change instead of build to last

So my primary guideline would be **don't even consider microservices unless you have a system that's too complex to manage as a monolith**. The majority of software systems should be built as a single monolithic application. Do pay attention to good modularity within that monolith, but don't try to separate it into separate services.
[16]

Other patterns

- Domain-driven development
- Layered architecture
- Model-view-controller

Domain Driven design (DDD)

- The software systems is build “around” the core domain knowledge and concepts of a business
- The structure and language of components, software code (class names, class methods, class variables) and data entities **should match the business domain**
  - Domain model and glossary must be followed
- Example:
  - Components: Product Catalog, Shopping Cart, Orders, …
  - Classes: Product, ShoppingCart, ShoppingCartItem,...
  - Methods: ShoppingCart -> addProduct, removeProduct, ..

Layered / tiered architecture

Layered architecture

- Components are organized into **horizontal layers**
- Each layer performs a specific role within the application (→ SoC)
- 3/4-layer architecture (logical separation in the code, but not necessarily physical):
  - Presentation / UI layer - handles user interaction and presentation
  - Application / business layer - coordinates use cases and workflows
  - Domain layer (optional) - business rules, concepts, domain logic
  - Data access layer - handles access to and persistence of stored data

Tiered architecture

- 3-tier architecture (separation at the deployment level):
  - Presentation tier - provides the user interface and handles user interaction
  - Application tier - processes application requests and executes application logic
  - Data tier - stores and manages data, typically using a database management system (DBMS)

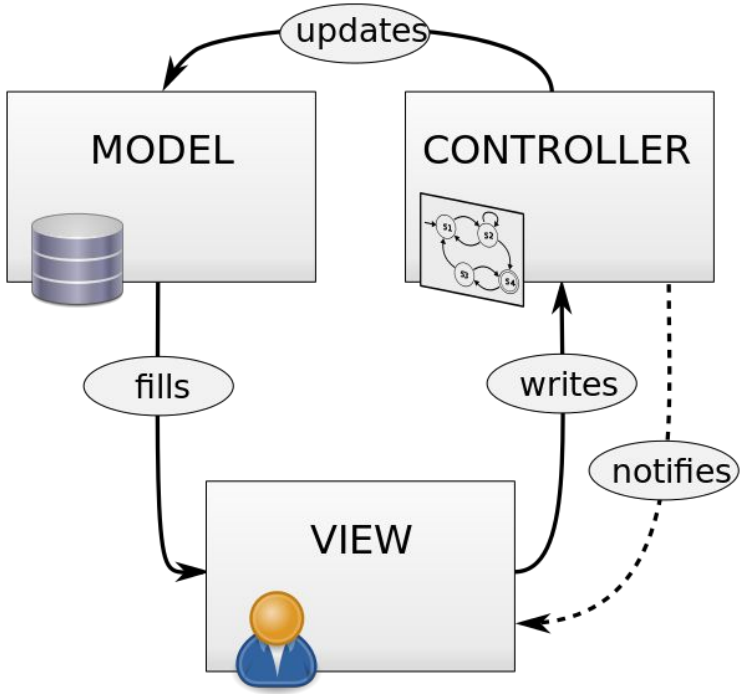
Example: Mapping in web application

- Presentation tier - web browser (presentation layer)
- Application tier - web servers / app servers (application + domain + data access layer)
- Data tier - database servers (outside the application’s logical layers)

Model-view-controller

= pattern used commonly for developing user interfaces

- **The model** manages the data of the application
  (= state, often represented by in-memory objects)
- **The view** renders presentation of the model in a particular format (chart, table, ..)
- **The controller** processes the user input and updates the data model objects.



Example: MVC in frontend

(Git) repository ecommerce-store-frontend

```
/src
  /assets      (static resources)
  /views       (React components and pages)
  /controllers (user-action handling and coordination
  /models      (application data and state)
  /api         (communication with backend)
```

- The original **`/services`** folder has been split into **`/controllers`** **`/models`** and **`/api`**
- The /api folder contains API clients and is an infrastructure component outside the core MVC pattern
- Since the application uses an SPA architecture, the backend typically returns JSON rather than rendered HTML. It may use controllers and models, but it does not have a traditional MVC View

Cloud service models

- Ready-to-use infrastructure and services
- Address many architectural concerns (scalability, reliability, availability, security, flexibility, global reach, ..)

IaaS (Infrastructure as a Service)

- Provider Manages: Hardware, networking, virtualization
- You manage: OS, runtime, application code, data

PaaS (Platform as a Service)

- Provider Manages: + OS, runtime
- You manage: Application code, data

SaaS (Software as a Service)

- Provider Manages: + Application code
- You manage: Data (& configurations)

Big Three:

- Amazon AWS
- Google Cloud
- Microsoft Azure

See for example
Google Cloud models

Resources

[1] OMG® Unified Modeling Language® (OMG UML®), Version 2.5.1, 2017

[2] SWEBOOK v3

[3] Ian Sommerville: Software Engineering (10th edition)

[4] Robert Lukotka: Architecture

[5] Martin Fowler: Design Stamina Hypothesis

[6] Wikipedia: Domain-driven design

[7] Philippe Kruchten: Architectural Blueprints - The “4+1” View Model of Software Architecture, 1995.

[8] MVC Diagram (Model-View-Controller) byXinfe, CC BY-SA 3.0

[9] Service Oriented Architecture : What Is SOA?, The Open Group SOA Working Group.

[10] Michal Kostič: Service-oriented architectural patterns

[11] Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, 2003.

[12] R. Červenka: UML Components

[13] The component diagram (IBM)

[14] Wikipedia: Global illustration of the 4+1 Architectural View Model

[15] Thomas Erl: Service-Oriented Architecture: Analysis & Design for Services and Microservices (Second Edition)

[16] Martin Fowler: Microservices, Microservice Premium