Microservices and other Software Architectures

dr. Rok Piltaver, siječanj 2024

FAKULTET INFORMATIKE I DIGITALNIH TEHNOLOGIJA
SVEUČILIŠTE U RIJECI

Software Architectures for Computer Applications

Monolith

Multi-tier architectures

Microservices

Serverless

VIdeos:

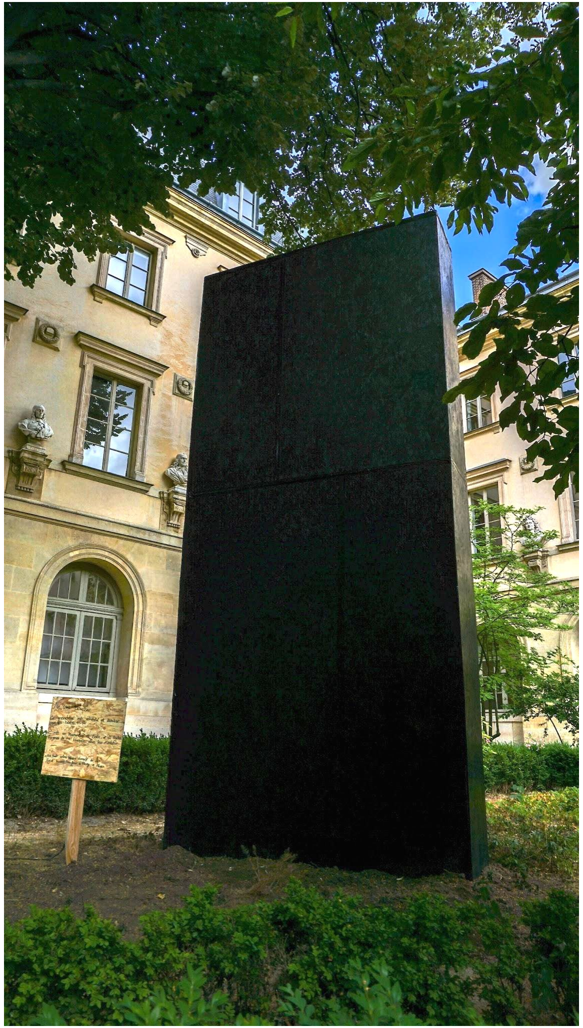
- Microservices Explained in 5 Minutes
- Microservices explained - the What, Why and How?

Monolith

Monolith

A single unified software application which is self-contained and independent from other applications.

- **Performs every step** to help the user carry out a complete task, end to end.
- Monolithic **software** is a **single**, large, and **cohesive unit.**
- Often associated with **mainframe computers & PCs**.
- **Examples**: personal finance apps, word processors, single-player games.



Monolith - Key Characteristics

- **Single Codebase**: entire apps (user interface, business logic, data access layers) is developed and maintained as a single codebase.
- **Development Simplicity**: monolithic architectures can be **simpler to develop initially**, as all the components are part of the same codebase.
- **Single Deployment Unit**: entire application is deployed as a single unit. **Updates** or changes to any part of the application **require deploying the entire** monolith.
- **Tight Integration**: all app components and modules are tightly coupled and interdependent - **changes to one part** of the system **may affect other parts**.
- **Scalability Challenges**: scaling can be challenging because the **entire app needs to be scaled**, even if only a specific component requires more resources.

Monolith - Disadvantages

As a monolithic app grows in complexity & size, it becomes difficult to:

- **maintain**: changes may affect other parts, build time, delivering updates, single language, shared dependencies.
- **scale**: entire app needs to be scaled, vertical scaling limits and cost
- **evolve**: typically lacks flexibility

Multitier architecture

Multitier architecture

A client–server architecture in which **presentation**, application **processing** and **data** management functions are physically **separated** (on separate platforms).

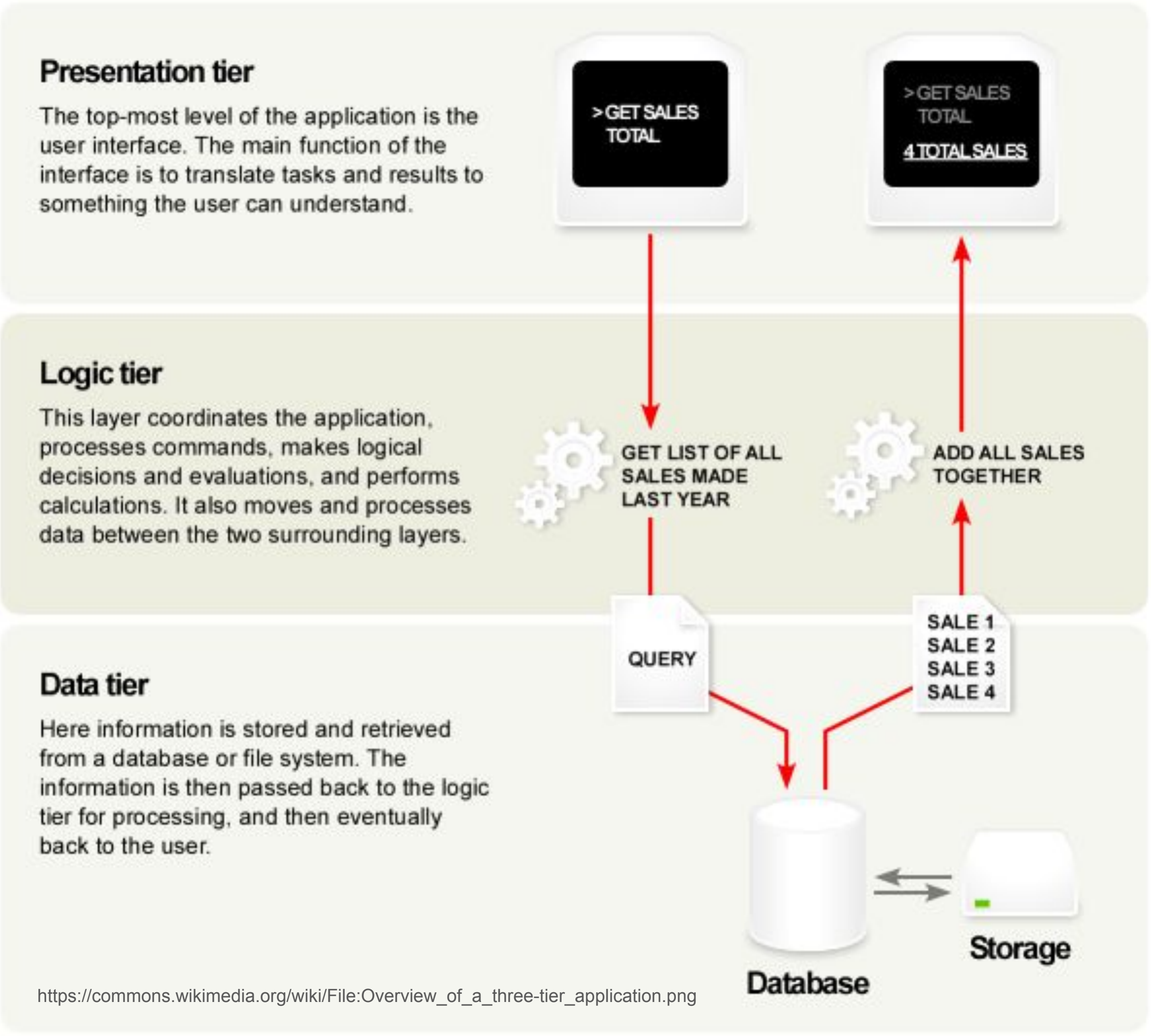
- Segregating app into **tiers** enables modifying or adding a tier, instead of reworking the entire app.
- Any **tier can be upgraded or replaced independently** in response to changes in requirements or technology.
- Tiers communication through **well-defined APIs**.
- Tiers often run on separate physical servers (each tier may run on a cluster).
- Apps become more **flexible** and **reusable**.

Three-tier Architecture

Developed and maintained as three independent modules/tiers/layers:

- **presentation**: displays information and can be directly accessed by user
  - runs on a PC or workstation
  - e.g. web page, OS’s GUI
- **business logic**: controls app’s functionality - performing processing logic.
  - runs on workstation or server
- **data tier**: persistent data storage and data access through an API
  - e.g. DB, file store

Three-tier Architecture



Presentation tier

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

https://commons.wikimedia.org/wiki/File:Overview_of_a_three-tier_application.png

Three-tier Architecture for Web Development

Front-end web server: servs static and cached dynamic content to be rendered in a browser.

Application server for dynamic content processing and generation
(e.g., Symfony, Spring, ASP.NET, Django, Rails, Node.js).

Back-end database (or data store), persists and provides access to the data.

Microservices

Microservices

App as a collection of loosely coupled (independent), fine-grained (small) services, communicating through lightweight protocols.

- Each service focuses on a specific business capability.
- Small and focused services **reduce dependencies** in the code base.
- Services **communicate over a network**.
- **Interfaces** need to be designed carefully and treated as a public API.
- Services **hide complexity** from their users.
- Allows for greater **flexibility, scalability, agility**, and easier maintenance.
- Enables developing SW with **fast growth and size**, and using off-the-shelf services easily.

Advantages of Microservices

- **Team independence:** teams work on separate microservices with independent dev cycles**.**
- **Flexibility and agility**: independent teams develop and deploy fast and frequently.
- **Continuous delivery**: facilitates automation of testing, deployment, and monitoring.
- **Technology diversity**: can choose the best technologies for each specific task.
- Easier **adoption of cloud services**: well-suited for cloud environments.
- **Scalability**: independent scaling of each services based on their specific needs.
- **Efficient resource utilization**: each service deployed on the appropriate infrastructure and scaled independently based on its load.
- **Fault Tolerance**: can be designed with redundancy and failover mechanisms.
- **Fault Isolation**: failed microservice doesn't bring down the entire system.

Challenges with Microservices

- **Increased complexity** of development, testing, deployment, and maintenance: handling service discovery, versioning and API compatibility, communication between services, error handling, and distributed data management.
- **Communication** over the network leads to potential **latency** and increased communication **overhead** (compared to in-process communication)
- **Maintaining consistency** across distributed data stores/DBs (all services having access to the most up-to-date data can lead to complex solutions (distributed transactions or other synchronization mechanisms).
- **Deployment and testing** challenges: ensuring compatibility between different versions, end-to-end testing in a distributed environment.

Challenges with Microservices

- Increased **operational overhead**: managing, troubleshooting, debugging, and monitoring a large number of distributed services can increase operational overhead.
- **Security challenges**: network security, data protection, access control, additional attack vectors. Implement robust security measures, including secure communication, proper authentication, and authorization mechanisms.
- **Initial development cost**: transition from a monolithic architecture involves significant upfront dev costs, including refactoring existing code, redesigning systems, and training teams on new technologies and practices.
- **Tooling and infrastructure**: implementing and maintaining tooling for CI, CD, and monitoring.

Serverless

Serverless Architecture

Serverless architecture (Function as a Service - FaaS) allows developers to run individual functions or pieces of code in response to events without managing the infrastructure.

- Functions are **stateless** and **event-triggered**.
- **Automatic scaling and resource allocation** by the cloud provider.
- **Billed** based on **actual usage** (e.g., execution time).

Advantages of Serverless Architecture

- **Scalability**: serverless platform automatically scales resources based on incoming requests.
- **Cost efficiency**: pay-as-you-go pricing model based on the actual execution time (good for variable and unpredictable workloads)
- **Reduced operational overhead**: serverless platform handles infrastructure provisioning and maintenance.
- **Rapid development and deployment**: development is often faster as developers can focus on writing small, independent functions. Deployment is simplified, enabling faster release cycles.
- **Event-driven model**: apps can easily connect and react to events (triggers) from different sources, enhancing flexibility and extensibility.
- **Infrastructure abstraction**: abstracts away the underlying infrastructure, devs do not have to manage servers, operating systems, or runtime environments.

Disadvantages of Serverless Architecture

- **Cold start latency**: when function is triggered after period of inactivity, there may be a delay due as the serverless platform initializes resources for the function.
- **Execution time limit**: functions have execution time limits imposed by the provider.
- **Limited resource control**: apps with specific resource requirements may face customization limitations (no control over memory, CPU, and networking)
- **Vendor lock-in**: moving between providers may involve rewriting code and adapting to platform-specific features.
- **Stateless nature**: functions must be stateless, maintaining state between function invocations requires external storage solutions.
- **Limited execution environment**: restrictions on the types of runtimes, libraries, or languages that can be used.