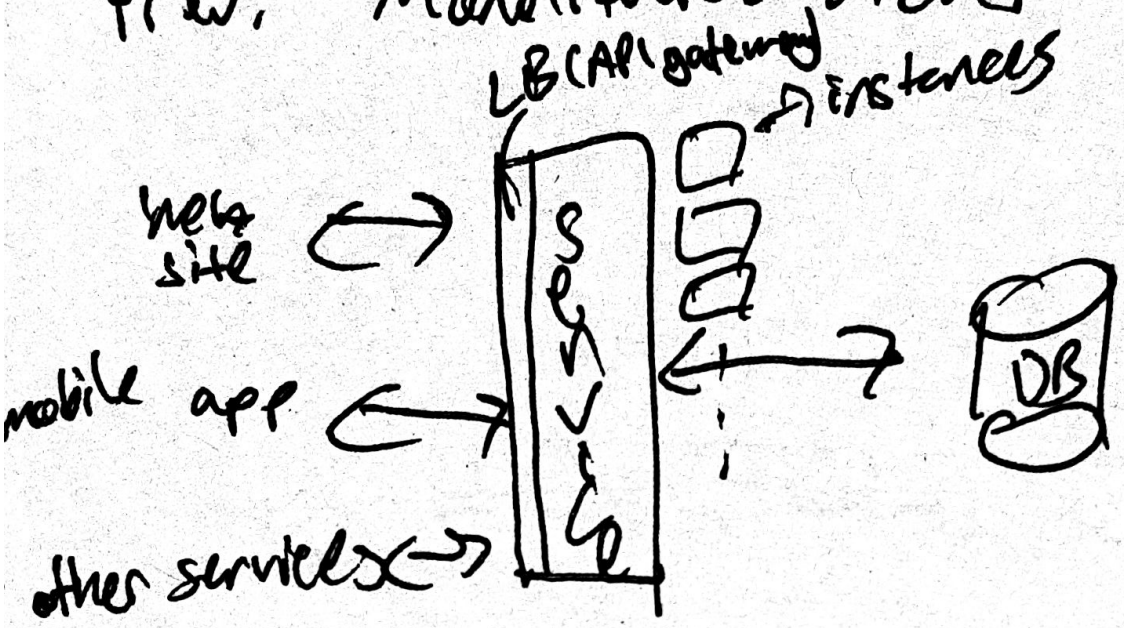
Training: Microservices Architecture

Prev. Monolithick arch.



Services Oriented Architecture

- reusability: Everyone uses same piece of code.
- no client upgrade & if signature of method stays same client doesn't need an upgrade on their contract
- standardized contracts (backward compatible)
- stateless: you don't need to remember prev. requests
- scalability: add more instances

Microservices extends SOA

- Micro sized service provide:
  - Efficiently scalabe apps
  - Flexible apps
  - High perf. apps
- Apps powered by multiple services
- Small service with single focus
- Lightweight communication mechanism
  client2service service2serv.
- Technology agnostic API (on common comm. protocol like HTTP REST)
  ex. .NET client can talk to Java microserv.
- Independent DBs (each has one)
- Services independently changable / deployable
- Distributed transactions
- Needs centralized management tool

Why now?

- Can change quickly
- More reliable
- Business domain-driven design
- Automated test tools (contract may happen in several services, need to test these automatically)
- Centralized release management tools
- On demand hosting (create new VM / clone)
- Online cloud services
- Embrace new tech. faster (can change the techstack service independent)

Benefits

- Shorter dev. times (smaller scopes)
- Reliable and faster deployment

Enables
- Frequent updates
- Decoupling changeable parts
- Security (no single point failure)
- Right technology (more effort for hacking)
- Increased uptime → (can detect which server has issue)
- Fast issue resolution
- Highly scalable & better performance
- Better know-how (teams can share services)

Microservices Design Principles

- High Cohesion →
  - Single focus & responsibility
    - SOLID principle (class can only change for one reason)
  - Encapsulation principle (OOP)
    (package all related data & functionality into one package
  - Easily rewriteable code
  - Why: - Scalability
    - Flexibility
    - Reliability

* - Autonomous (work)
  - Loose coupling (decoupling) between microservices
    (change in service should not force other services to change)
  - Honour contracts and interfaces (shared model)
  - Stateless
  - Independently changeable / deployable

- Backwards compatible
- Concurrent development ← ownership / ← versioning

- Business Domain Centric

- A service represents business: function / domain
- Scope of service
- Bounded context from DDD
- Identify boundaries (also fix incorrect ones) → split/merge
- Shuffle code if required → a function may relate to two services
  - group related code into a service
  - aim for high cohesion
- Responsive to business change

- Resilience (zorluklara tepki)

- Embrace failure
  - another service
  - specific connection
  - 3rd party system

↳ key degrading functionality (in case of missing service.)
- default func.
- multiple instance (register on startup / deregister on failure)

- Design for known failures

- Types of failure
  - Exceptions/Errors
  - Delays
  - Unavailability
- Network issues
  - Delay
- Validate input
  - service2service
  - [redacted]client2serv

- System Being Observable
  - System health
    - status
    - logs
    - errors
  - Centralized monitoring & logging (monitor everything in one place)
    - Why
      - distributed transaction
      - quick problem solving
      - quick deployment requires feedback
      - data used for
        - capacity planning
        - scaling
      - what's actually used
      - monitor business data

- Automation

- Tools to reduce testing
  - time taken on testing integration
  - environment setup for testing
- Tools to provide quick feedback
  - Cont. Integration
- Tools to provide quick deployment
  - Pipeline to deployment
  - Deployment ready status
  - Automated & reliable
  - Cont. Deployment
- Why - distributed system
  - multiple instances of services
  - manual integration testing too time consum.
  - manual deployment time consum. & inreliable

-Microservices Design

- **High cohesion** (continuously)
  - Question in code/peer reviews
    (if new microservice has one reason to change
  - Split until they have one reason to change
- **Autonomous**
  - Loosely coupled
    - Communication by network
      - Synchronous
      - Asynchronous
        Publish events
        Subscribe to events
        [via message broker]
    - Technology agnostic API
    - Avoid client libraries (increases coupling)
    - Avoid chatty exchanges between services & avoid sharing (be stateless)
      [db, shared libraries etc..]

- Versioning (must be sure supporting backwards comp.)
  - Avoid breaking clients (backwards compability)
  - Integration tests (test input&output of contracts)
  - Have versioning strategy
    - Concurrent versions (coexisting endpoints) (old&new) → slowly
  - Semantic versioning
    - Major. - not backwards comp.
    - Minor. - backwards comp.
    - Patch. - defect fix & back. comp.

- **Resilience**
  - Design system to fail fast
  - Use timeouts (for connected systems)
    - network outages
    - latency
    - assume if failed & deterministic
    - degrade / default behaviour
  - Monitor & log timeouts *

- Observable

- Centralized Monitoring
  - Monitor the host { CPU, memory, disk }
  - Expose metrics within the services
    - response times
    - timeouts
    - exceptions & errors
  - Business data related metrics (trends)

- Centralized Logging
  - When to log
    - Startup & shutdown
    - Code path milestones { requests, responses, decisions }
    - Timeouts, exceptions, errors
  - Structured logging
    - Log level
    - Date & time
    - Correlation ID (distributed between services) → traceable transaction between services
    - Hostname & service name & service instance
    - Message

• Automation

- CI Tools
  - Work with source control systems (automatic after check-in)
  - Unit tests & integration tests are required
    - test unit of code (ex. class, method)
    - between software modules (ex. input output)
  - Ensure quality of check in
    - Code compiles
    - Test pass
    - Changes integrate
    - Quick feedback → Urgency to fix quickly
- CD Tools
  - configure once
    - where from take the build
    - where to
    - time
  - Deployable after check-in
  - Benefits
    - Quick to market
    - Reliable deployment
    - Better customer experience

- Technology for Microservices

- Communication: Synchronous
  - Remote procedure call (RPC)
    (make a request with RPC library)
    hides the network details
  - HTTP (firewall friendly)
  - REST (CRUD using HTTP verbs
    Natural decoupling (does not
    dictate endpoint technology
    uses JSON/XML
    HATEOS: Includes additional
    links in response)
  - Issues
    - Both parties have to be available
      & visit each other
    - Client must know location
      of service (host/port)
    - Performance subject to network & availability

- **Asynchronous**
  - Event based
  - Decouples client & service
  - Message queueing protocol (brokers)
  - Subscriber & publisher are decoupled
  - Issues
    - Complicated in distributed transactions
    - Reliance on message broker [redacted] (another point of failure)
    - Visibility of the transaction
    - Managing message queues

[redacted] IRL we use both flavors

- Hosting Platforms

* Virtualization
  - Acts as a physical machine
  - Foundation of cloud platforms (Platform as a Service) ex. AWS
  - Could be more efficient
    - Takes time to setup / load
      quite bit of resources
  - Unique features
    - Take snapshot
    - Clone instances
  - Standardised and mature

- Containers (Type of virtualization)
  - They do not run entire OS within the container, they run minimal amount required to run the service.
  - Isolate services from each other
  - Single service per container

- Difference to a VM
  - Uses less resource
  - Faster
  - Quicker to create new instances
- Future of hosted apps
- Not established as VM
  - Not standardised
  - Limited features & tooling
  - Infrastructure support in its infancy
  - Complex to setup

- Registration
  - New instances register themselves on startup, deregister if fail
- Discovery
  - Client side (connects directly) (service registry db)
  - Server side (gateway, lb to connect on behalf of client)

- Centralized Monitoring & Logging

- Microservices Performance

• Scaling — Automated
On-demand

• Caching [redacted]
- reduces client calls to service
service " " db
service " " service

- API gateway level
- Client side level
- Service level
- Consideration
  • Simple to setup and manage
  • Data leaks

• API Gateway
- Helps with performance — load balancing
caching
- Help with — creating single entry point
exposing services to clients
one interface to many serv.

routing to specific instance of service
can use service registry db

* Security — Dedicated security service
central security vs. service level

- Automation Tools: CI

• Desired features — cross platform
source control integra.
notifications

*. Map microservice to a CI build
(code change only triggers specific serv.)
- Builds and tests run quicker
- Seperate code repository for service

*. Avoid one CI for all services

- CD

• Aim for cross platform
• Desired features — central control panel
support for scripting
integration with CI

- support for multiple environments
- support for PAAS

- Brownfield Microservices (migrating already existing app. to microserv.)

* Approach
  - Existing system ← monolithic, organically grown, seems too large to split
  - Lacks ms. design principles
  - Identify seams — separation that reflects domains; identify bounded contexts
  - Start modularizing bounded contexts
    - move code incrementally (start off with one)
    - tidy up a section per release
    - run unit tests & integration tests

- Seams are future ms. boundaries

- Greenfield Microservices (new project)

- Start off with monolithic design
  - High level
  - Evolving seams
  - Develop areas into modules
  - Boundaries start to become clearer
  - Refine and refactor design
  - Split further when required
- Modules become services
- Sharable code libraries promote to service
- Review vs. principles at each stage
- Prioritise by [minimal viable product / customer needs & demands]