v0.1

MICROSERVICES

MESOSPHERE

AGENDA

1. Overview
2. Design Principles
3. Deployment
4. 12 Factor App
5. Modular Containers

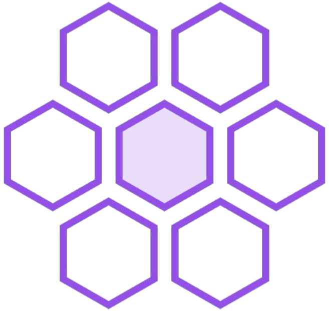
Microservices

OVERVIEW

OVERVIEW

noun | *ˈmīkrō/ /ˈsərvəs/* :

an approach to application development in which a large application is built as a suite of modular services. Each module supports a specific business goal and uses a simple, well-defined interface to communicate with other modules.*

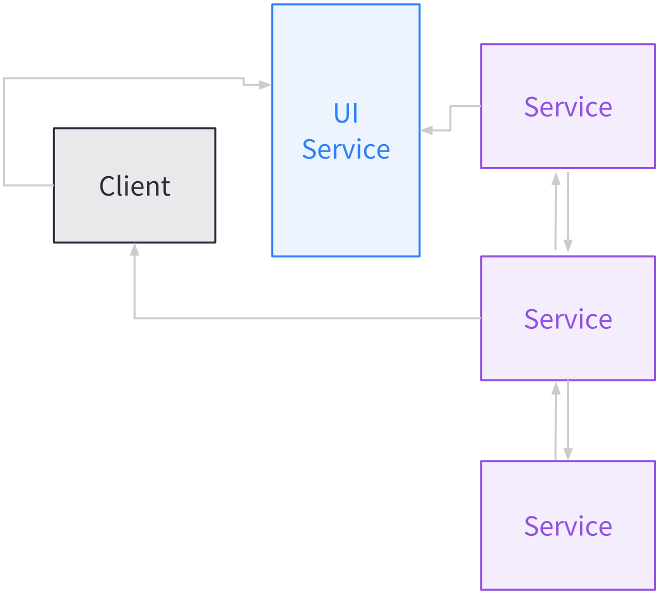


Microservices are designed to be **flexible**, **resilient**, **efficient, robust**, and **scalable**.

*From whatis.com

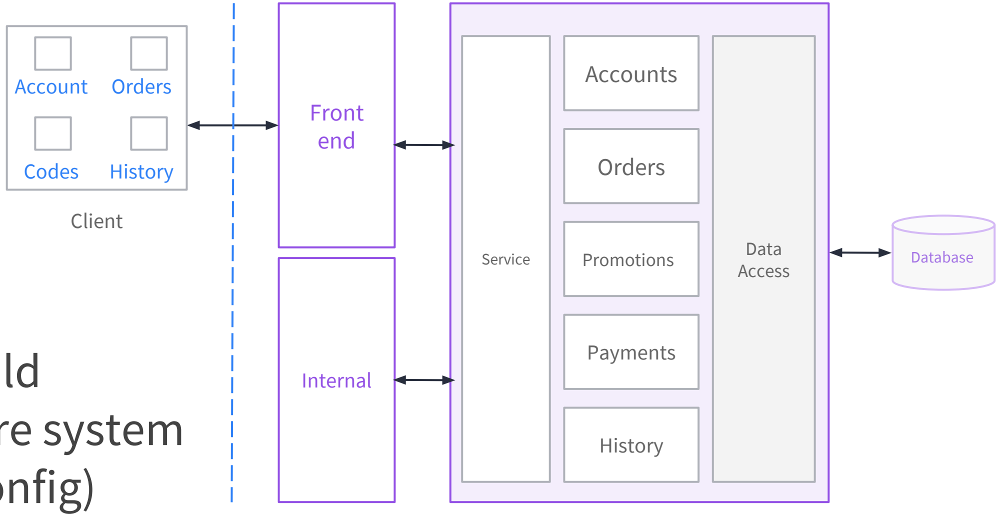
OVERVIEW

- Micro sized services
  - Efficiently scalable applications
  - Flexible applications
  - High performance
- Application(s) powered by small services with a single focus
- Lightweight communication mechanism
- Technology agnostic API
- Independently changeable and deployable



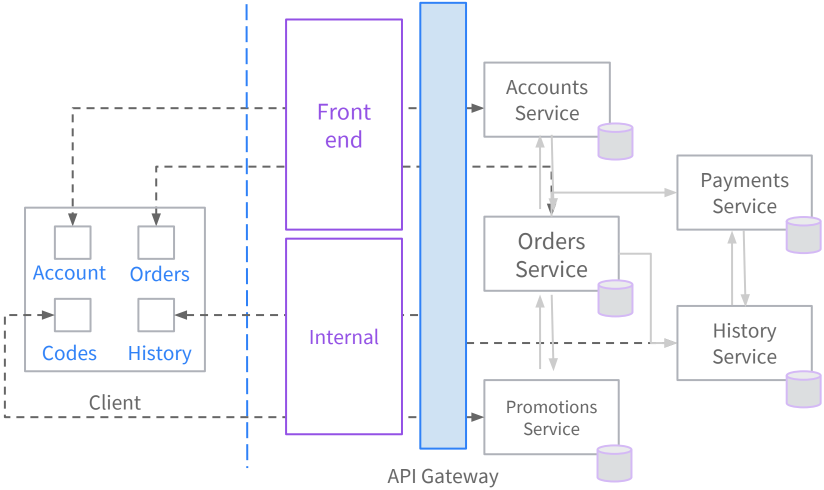
MONOLITHIC

- Challenges
  - No size restriction
  - Large codebase
  - Long development times
  - Inaccessible features
  - Highly coupled
  - Difficult to understand
- Failure affects whole system
- Changes may cause complete rebuild
- Scaling requires duplication of entire system
- Fixed technology stack (complex config)
- Easier to replicate (self-contained)

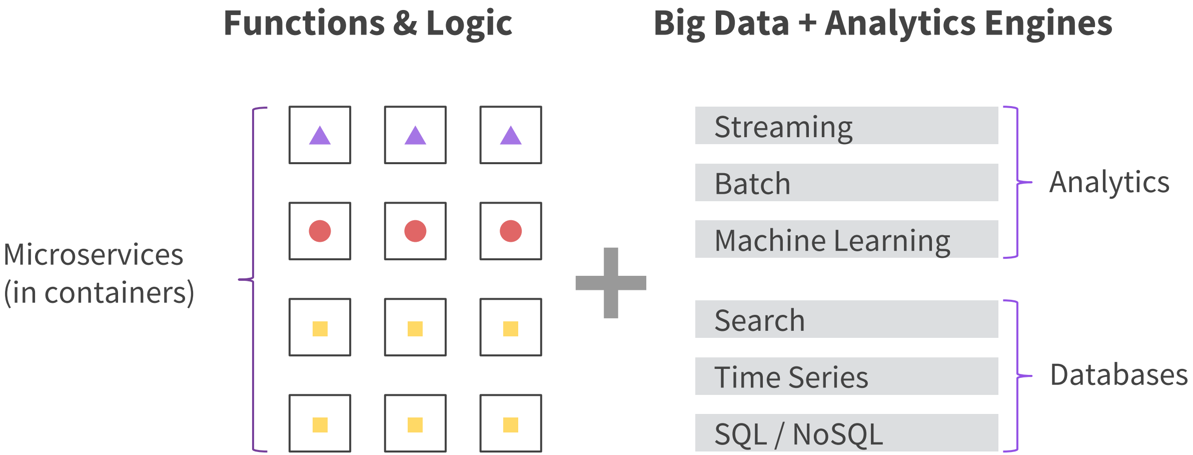


MICROSERVICES

- Benefits
    - Shorter development times
    - Decoupled
    - Increased uptime
    - High scalable
    - Tested individually
    - Simpler to understand
- Enables distributed teams
- Right technology
- Automated test tools
- Faster innovation, time to market



THE MODERN ENTERPRISE APPLICATION



- Microservices
- Containers
- Stateful Big Data
- Open source

Microservices

DESIGN PRINCIPLES

DESIGN PRINCIPLES

- **High Cohesion**: Single focus done well
- **Autonomous**: Independently changeable and deployable
- **Business Domain Centric**: Represents business function or domain
- **Resilience**: Embrace failure
- **Observable**: Centralized logging and monitoring
- **Automation**: Tools for testing and deployment

DESIGN PRINCIPLES

- **High Cohesion**
  - Split service until it has a singular focus, only one reason to change
- **Autonomous**
  - Loosely coupled, versioning strategy, ownership by team
- **Business Domain Centric**
  - Identify business domains and subdivide into functions
- **Resilience**
  - Design for failures, fail fast and recover fast
- **Observable**
  - Tools for centralized monitoring and logging
- **Automation**
  - Continuous Integration and continuous development tools

DESIGN CONSIDERATIONS

- Communication
  - Synchronous or Asynchronous
- Hosting platform
  - Cloud, on-prem, virtual machines, containers
- Service Discovery
- Monitoring and Logging Tools
- Performance
  - Auto scaling, caching, load balancing
  - Tools for centralized monitoring and logging
- Automation
  - Continuous Integration and continuous development tools

Microservices

DEPLOYMENTS

DEPLOYMENT TYPES

- Brownfield
  - Existing monolithic system
  - Lacks microservices design principles
- Greenfield
  - New project
  - Evolving requirements
  - First microservice

BROWNFIELD APPROACH

- Identify seams
  - Separation that reflects domains
  - Identify bounded contexts
  - Seams are future microservice boundaries
- Modularize bounded contexts
  - Move code incrementally
  - Keep existing functionality intact
  - Continuous unit testing and integration tests to validate

BROWNFIELD MIGRATION

- Code organized into business domain or function
- Convert bounded contexts into microservices
    - Start with the easiest one
    - Maintain both versions for easy rollback
- Prioritize what to split by risk, technology, dependences
- Iterate incrementally
- Validate integration with monolithic
- Avoid shared databases, split using seams
- Consolidate reporting across microservices

GREENFIELD APPROACH

- Start with monolithic design
  - High level picture
  - Evolving seams
  - Develop areas into modules
  - Boundaries start to develop
  - Refine and refactor design
- Eventually turn modules and shareable code libraries into services
- Review principles at each stage
- Map out target state and prioritize activities accordingly

GREENFIELD CONSIDERATIONS

- Initially there will be:
  - Longer development times
  - Cost and training for tools and skills
  - Addition testing
  - Cost of improving infrastructure
  - Cloud technologies
  - Culture change

Microservices

12 FACTOR APP

12 FACTOR APP

Methodology for building software-as-a-service apps that:

- Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project
- Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments
- Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration
- **Minimize divergence** between development and production, enabling **continuous deployment** for maximum agility
- And can **scale up** without significant changes to tooling, architecture, or development practices

From http://12factor.net/

12 FACTOR APP

I. Codebase

- One codebase tracked in revision control, many deploys

II. Dependencies

- Explicitly declare and isolate dependencies

III. Config

- Store config in the environment

IV. Backing Services

- Treat backing services as attached resources

From http://12factor.net/

12 FACTOR APP

V. Build, release, run

- Strictly separate build and run stages

VI. Processes

- Execute the app as one or more stateless processes

VII. Port binding

- Export services via port binding

VIII. Concurrency

- Scale out via the process model

From http://12factor.net/

12 FACTOR APP

IX. Disposability

- Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity

- Keep development, staging, and production as similar as possible

XI. Logs

- Treat logs as event streams (log to stdout & stderr)

XII. Admin processes

- Run admin/management tasks as one-off processes

From http://12factor.net/

Microservices

MODULAR CONTAINERS

MODULAR CONTAINER

- Proper Linux process
  - React to signals, return proper exit codes, use standard streams
- Explicit interfaces
  - Make dependencies explicit (CLI args, env vars, labels)
- Disposable
  - Keep ephemeral state, robust against failure, minimal setup
- Immutable
  - Changes to container should be made by rebuilding
- Self-contained
  - Zero-config deploy, add dependencies at build time, dynamic configs
- Small
  - Minimize amount of code, use small base image

SUMMARY

In this module we looked at microservices

- Monolithic applications to microservices
- Design principles
- Deployment considerations
- 12 factor app methodology
- Building modular containers

MESOSPHERE