Microservices Architecture

Outline

- Microservices architecture
  - Overview
  - Characteristics
  - Development team considerations
- Design considerations
  - Components & coordination
  - Communication
  - Data distribution & sharing
- Examples

Microservices architecture

Architectural style: server implemented as set of interacting **microservices**

- Small **stateless** services
- Each has single responsibility
- Each service has own API
- Communicate by exchanging messages

Stateless software service

- Software component accessible over Internet
- Does not maintain any internal state
  - Given input, produces corresponding output without side effects
  - Any state information is either…
    - Stored in a database service attaches to, or
    - Maintained by the service requestor (user/caller)
      ➔Passed with request, returned as part of result

Component decomposition

component ➔ service

Critical because…

- Benefits of cloud-based software (scalability, reliability, & elasticity) require components that
  - Can be easily replicated
  - Run in parallel (potentially across multiple servers)
  - Can be moved between virtual servers
- Components can be developed by different teams
- Components can be reworked/replaced when underlying technology changes

Advantages of Microservices Arch

Addresses two problems with multi-tier software architecture for distributed systems

- **Development and deployment**
  - Each microservice deployed in own container, so can stop/restart without affecting other parts of the system
  - Monolithic architecture requires whole system reset
- **Scaling**
  - As demand increases, service replicas can be spun up
  - Monolithic architecture requires whole system to scale (even if demand is localized to a few components)

Outline

- Microservices architecture
  - Overview
  - Characteristics
  - Development team considerations
- Design considerations
  - Components & coordination
  - Communication
  - Data distribution & sharing
- Examples

Goal: Business-oriented

- A service should be related to a single **business function**/process/method
  - Collection of related, structured activities/tasks that produces a service or product for particular customer(s)

What are some business functions for your project?

- Ideally, focused on customer need, rather than providing technical service
  - Technical service could be imported as library

Goal: Lightweight

- Minimal communication overhead
- Rule of twos:
  - Microservice can be developed, tested, deployed by development **team** in ≤ 2 weeks
  - **Team** size: whole team can be fed by no more than two large pizzas (Amazon's guideline)
    - lower limit: **3-4 people** [team size for CS518, Capstone]
    - upper limit: **8 to 10 people**

Goal: Independence

- Self-contained
  - Uses own DB (not shared DB or other service)
  - Manages own user interface
- Implementation independent
  - Services interact via APIs—Programming language, other implementation details are hidden
- Independently deployable
  - Can replace/replicate service without changing other services in the system

Goal: low coupling, high cohesion

- **Coupling**: # of relationships a component has with other components in the system
  - Low coupling: component does not have many relationships with other (external) components
- **Cohesion**: # of relationships that **parts** of a component have with each other
  - High cohesion: all (most) parts needed to deliver the functionality are included in the component
  - Can also have to do with conceptual cohesion

Outline

- Microservices architecture
  - Overview
  - Characteristics
  - Development team considerations
- Design considerations
  - Components & coordination
  - Communication
  - Data distribution & sharing
- Examples

DevOps or DevSecOps

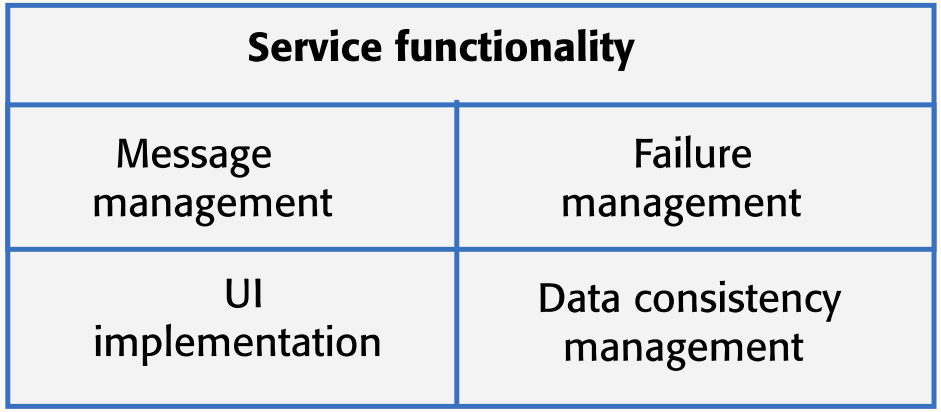
- Team responsible for more than functionality
  - Must also develop all the code necessary to ensure that a microservice is completely independent: UI code, security code, etc.
- Team also responsible for testing, operations, support, etc.
- (More on this topic in a later module)

Support code

Key code required for complete microservice

- Failure mgmt. for when…
  - microservice can't complete requested operation
  - external service returns error or does not reply
- Data consistency mgmt.
  - Communicating data updates between services

Microservice X



Outline

- Microservices architecture
  - Overview
  - Characteristics
  - Development team considerations
- Design considerations
  - Components & coordination
  - Communication
  - Data distribution & sharing
- Examples

Composition

What microservices will make up the system?

- Too many small services lead to…
  - Increased communication between services
  - Increased processing time because each service has to bundle/unbundle services sent from other services
- Too few larger services lead to…
  - Larger services with more dependencies
  - Makes changing services more difficult

Composition guidelines

- Balance fine-grain functionality with system performance
- Common closure principle:
  - Elements likely to change at the same time should go in the same service
- Associate services with business capabilities
- Give services access only to data they need
  - Minimize propagation of change in data

Coordination

How should these services work together to achieve desired results?

- Workflow could be **orchestrated**
  - Have a service that determines which system components should be active
  - Easier to debug
- Alternative approach: **choreography**
  - Use publish-subscribe model with services sending each other events
  - Leads to lower coupling, but can be tricky to get right

Failure Management

- How should failures be detected?
  - Could use timeout—fail if too much time passes
  - Could use “circuit breaker”
    - Forwards successful responses immediately
    - If a service isn’t responding, circuit breaker “trips”, and future requests immediately get error response
- How are failures reported and managed?
  - HTTP response codes, logs, etc.

Outline

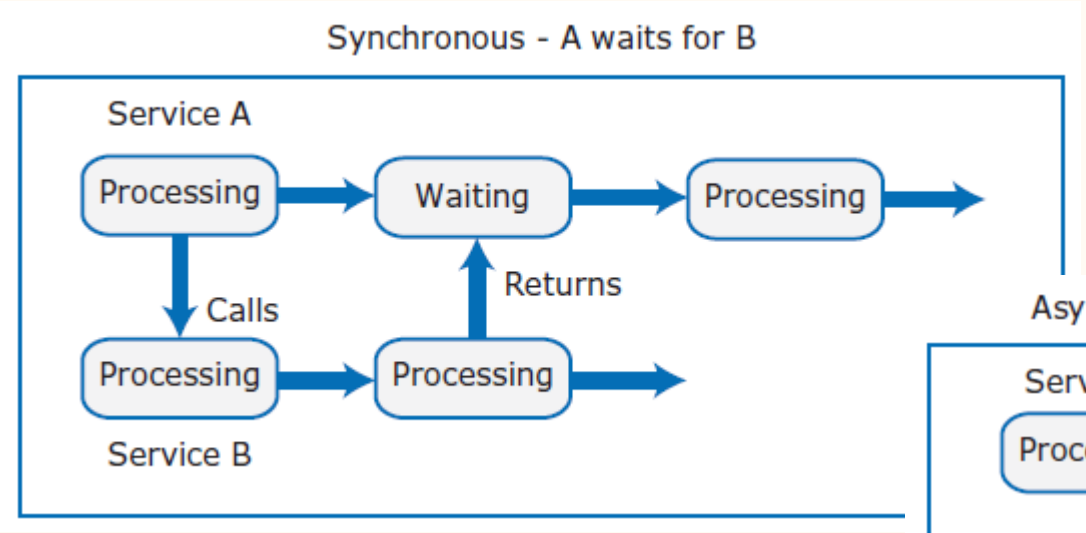
- Microservices architecture
  - Overview
  - Characteristics
  - Development team considerations
- Design considerations
  - Components & coordination
  - Communication
  - Data distribution & sharing
- Examples

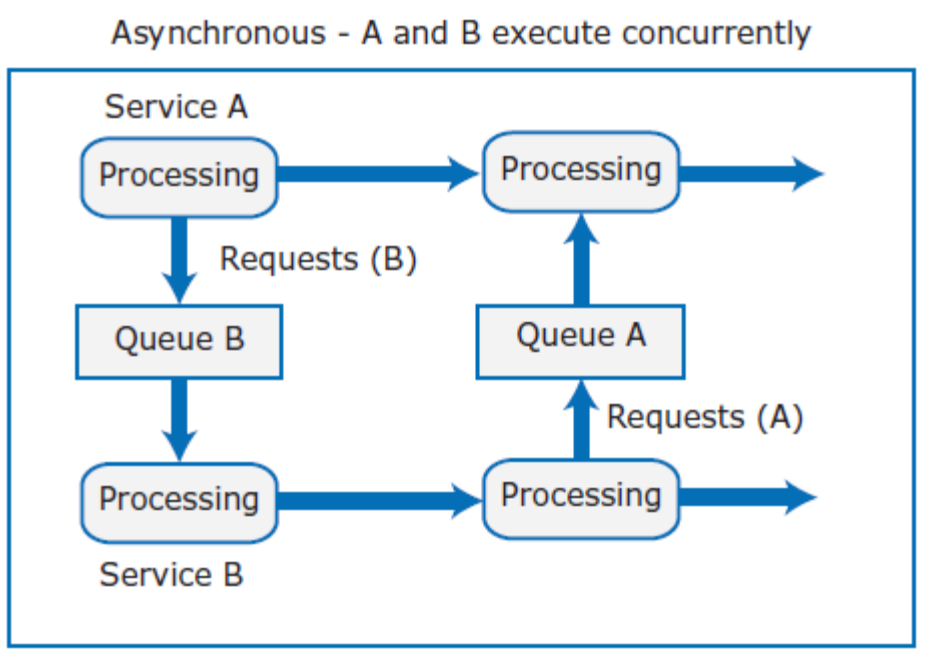
Communication

How should components communicate?

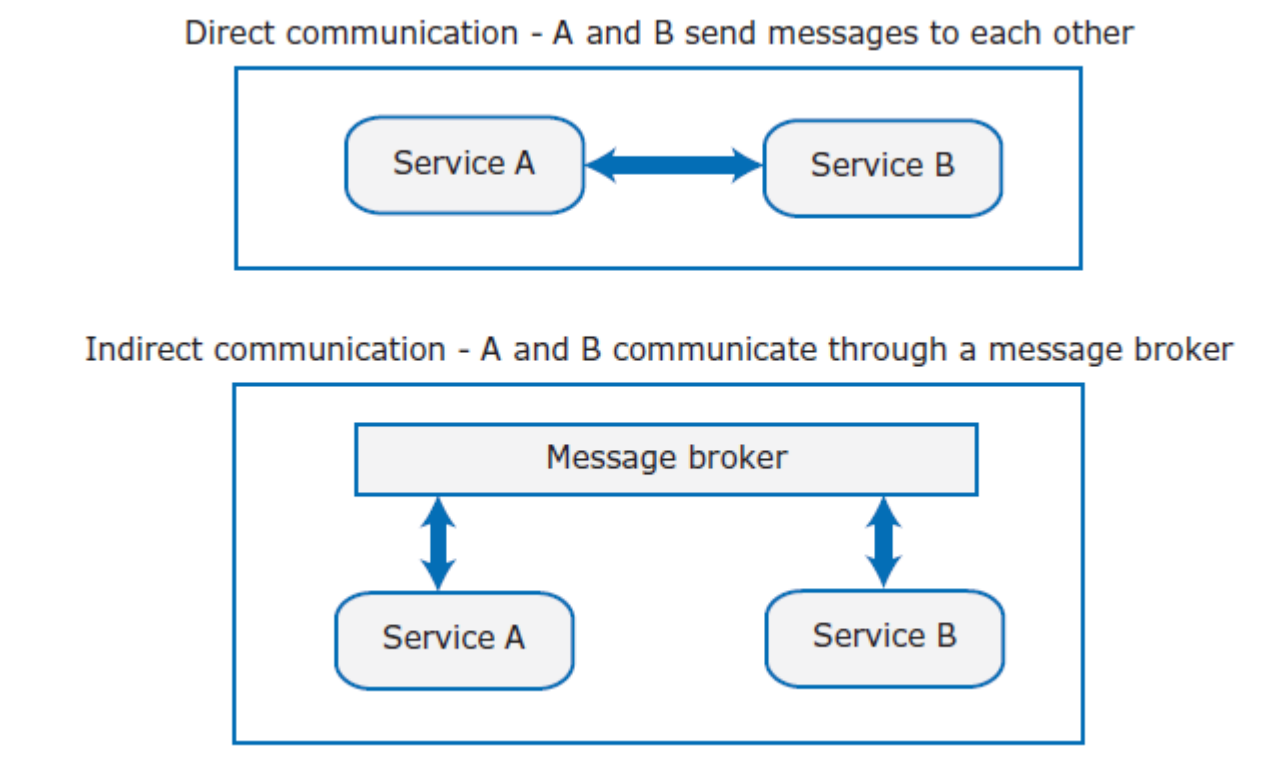
- Should service interaction be synchronous or asynchronous?
- Should services communicate directly or via message broker middleware?
- What protocol should be used for messages exchanged between services?

Synchronous vs. Asynchronous





Direct vs. indirect communication



Message protocol

- An agreement between services: how messages between services should be structured
- RESTful services follow the REST architectural style, often using JSON for message data
  - Service operations often represented using the HTTP verbs GET, PUT, POST, and DELETE
  - Service represented as resource that has own URI

Outline

- Microservices architecture
  - Overview
  - Characteristics
  - Development team considerations
- Design considerations
  - Components & coordination
  - Communication
  - Data distribution & sharing
- Examples

Shared database systems (recap)

Multi-tier client–server systems use shared database

- Access managed by a database management system (DBMS) which ensures that
  - Data always consistent
  - Concurrent data updates don't interfere w/ each other
- If implementing a system where *absolute* data consistency is a critical requirement (such as in banking) should use shared database architecture

Data independence & consistency

- Data independence
  - In theory: each service should manage own data
  - Reality: total data independence is impossible
- Data consistency
  - Systems using microservices must be designed to tolerate some degree of data inconsistency
  - Need a means to ensure that the common data are *eventually* made consistent

Types of data inconsistency

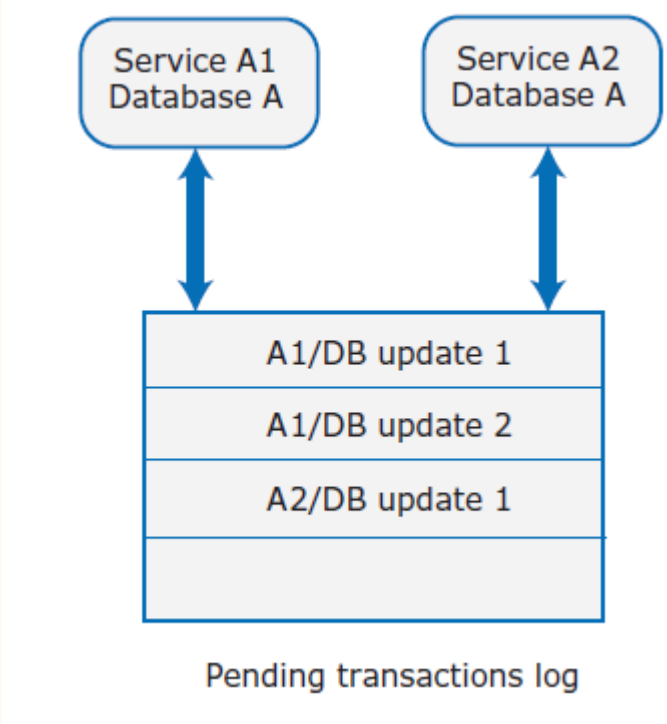
- **Dependent data inconsistency**
  - Actions or failures of one service can cause data managed by another service to become inconsistent
- **Replica inconsistency**
  - Several replicas of the same service may be executing concurrently each making updated to own copy of database

Ensuring data consistency

- Isolate data within each system service with as little data sharing as possible
- If data sharing is unavoidable design microservices so that
  - Most sharing is read-only
  - Keep # of services responsible for data updates low
- If services replicated, include mechanism that keeps database copies consistent

Eventual consistency

- System guarantees that the databases will have same data after a certain period of time
- Can be implemented by maintaining a **transaction log**

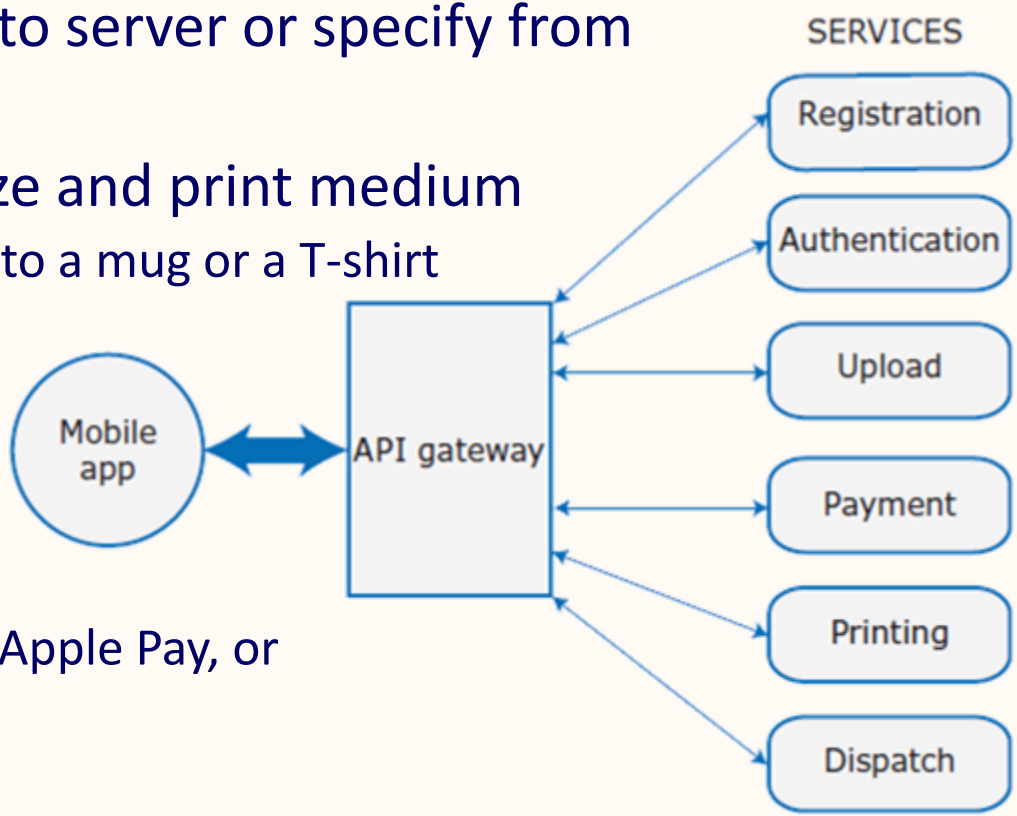


Outline

- Microservices architecture
  - Overview
  - Characteristics
  - Development team considerations
- Design considerations
  - Components & coordination
  - Communication
  - Data distribution & sharing
- Examples

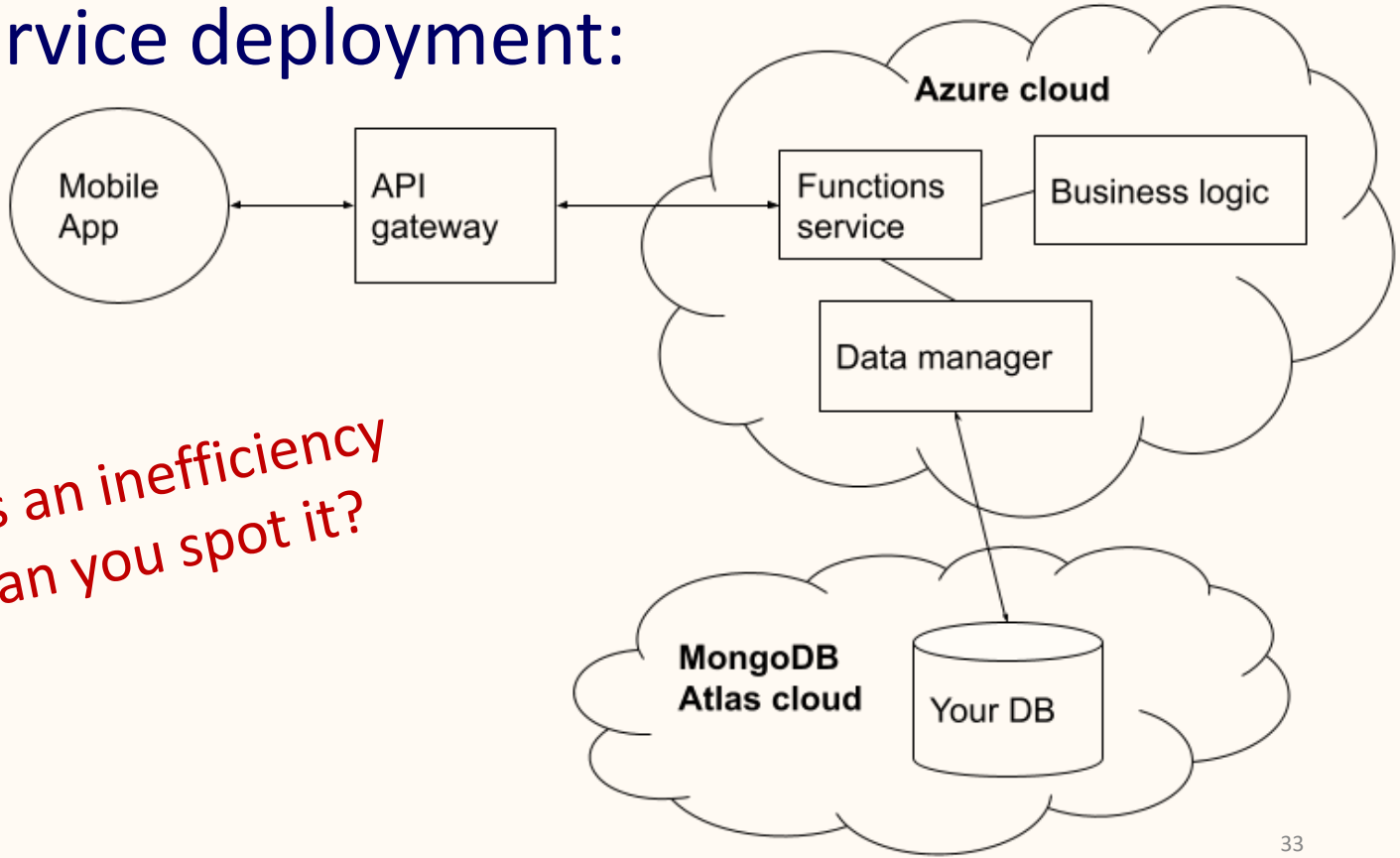
Photo printing system

- Users can upload photos to server or specify from Instagram
- Users can choose print size and print medium
  - Example: print a picture onto a mug or a T-shirt
- Prints are prepared and then mailed to them
- Payment happens through either by
  - Service such as Android or Apple Pay, or
  - Registering a credit card



Updated project architecture

- After service deployment:



There is an inefficiency here, can you spot it?

Outline

- Microservices architecture
  - Overview
  - Characteristics
  - Development team considerations
- Design considerations
  - Components & coordination
  - Communication
  - Data distribution & sharing
- Examples