CSCI427/927 Systems Development

Service-oriented architectural patterns
Microservices

Acknowledgement: Materials on these slides are adapted from "Engineering Software Products: An Introduction to Modern Software Engineering" by Ian Sommerville

Software services

- A software service is a software component that can be accessed remotely.
  - **Decompose** a system into **components** that can be easily replicated, run in parallel, and moved
  - The service is accessed through its published **interface** and all details of the service implementation are **hidden**.
  - Services often **do not maintain any internal state**. State information is either stored in a database or is maintained by the service requestor.
- When a service request is made, the state information may be included as part of the request and the updated state information is returned as part of the service result.
- As there is **no local state**, services can be dynamically reallocated from one virtual server to another and replicated across several servers.

Weather Forecasting Service

- This service can be accessed remotely by any client application, like a mobile weather app or a website, via the internet.
  - Easily replicated, run in parallel, and moved
  - Published Interface and Encapsulation
  - Stateless (does not maintain any internal state between requests and any necessary state information, like a user's location or preferences, is included in the request)
  - Any persistent state information must be managed by the client (service requestor) or stored in an external database
  - When experiencing high load, the service can be replicated to additional servers to distribute the load

Modern web services

- Big Web Services emerged in the early 2000s
  - World Wide Web during the late 1990s and early 2000s
  - Increased Demand for B2B Integration
  - Services exchanging large and complex XML texts
  - Time-consuming to analyze the XML messages and extract the encoded data
- Most software services do not require the full generality provided by these web service protocols.
  - services with low latency and high throughput
  - services with straightforward requirements, such as standard HTTP methods (GET, POST, PUT, DELETE) and typically using JSON
- Consequently, modern service-oriented systems, use simpler, 'lighter weight' service-interaction protocols that have lower overheads and, consequently, faster execution.

Microservices

- Microservices are **small-scale**, **stateless**, services that have a **single responsibility (one specific function or task)**. They are combined to create applications.
- They are completely **independent** with their own **database** and UI **management code**.
- Software products that use microservices have a *microservices architecture*.
- If you need to create cloud-based software products that are adaptable, scalable and resilient => design them around a microservices architecture.

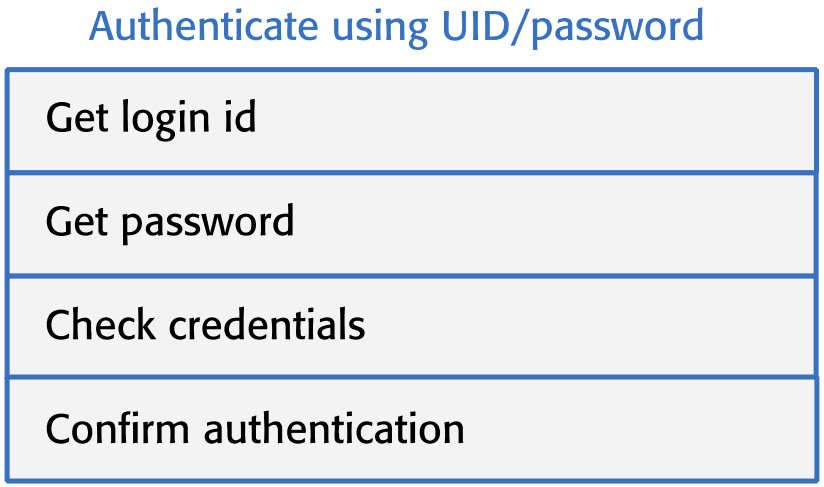
Microservice communication

- Microservices communicate by **exchanging messages**.
- A message that is sent between services includes **some administrative information**, a **request** and **the data** required to deliver the requested service.
- Services return a **response** to service request messages.
  - An authentication service may send a message to a login service that includes the name input by the user.
  - The response may be a token associated with a valid user name or might be an error saying that there is no registered user.

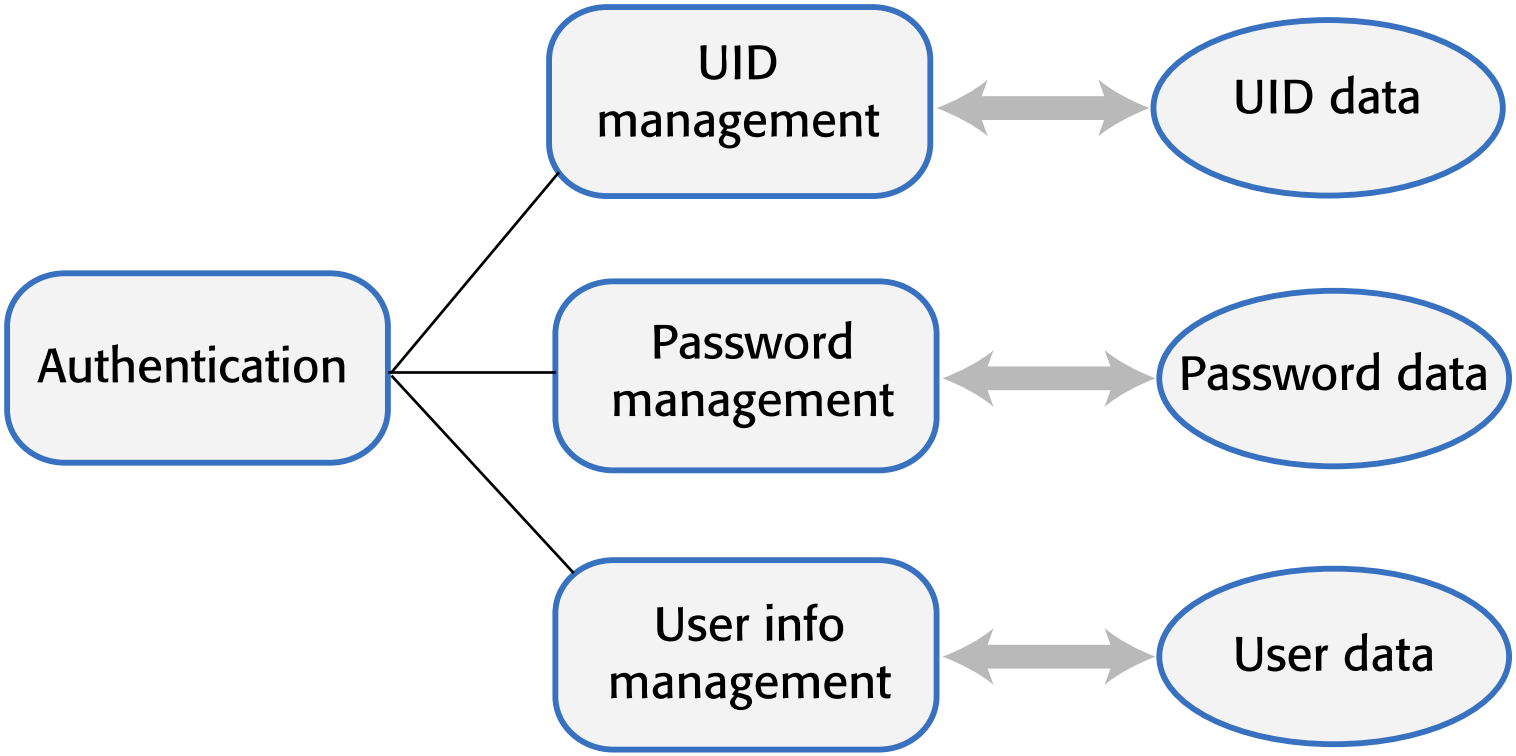
A microservice example

- Authentication system
  - User registration, where users provide information about their identity, security information, mobile (cell) phone number and email address.
  - Authentication using UID/password.
  - Two-factor authentication using code sent to mobile phone.
  - User information management e.g. change password or mobile phone number.
  - Reset forgotten password.
- Each of these features **could be implemented as a separate service** that uses a **central shared database** to hold authentication information.
- However, these features are too large to be **microservices**.
  - need to break down the coarse-grain features into more detailed functions.

Functional breakdown of authentication features



Authentication microservices



Characteristics of microservices

- ***Self-contained***
  Microservices do not have external **dependencies**. They manage their own **data** and implement their own **user interface**.

- ***Lightweight***
  Microservices communicate using **lightweight protocols** (https, websockets), so that service communication overheads are low.

- ***Implementation-independent***
  Microservices may be **implemented** using different programming languages and may use different technologies (e.g. different types of database) in their implementation.

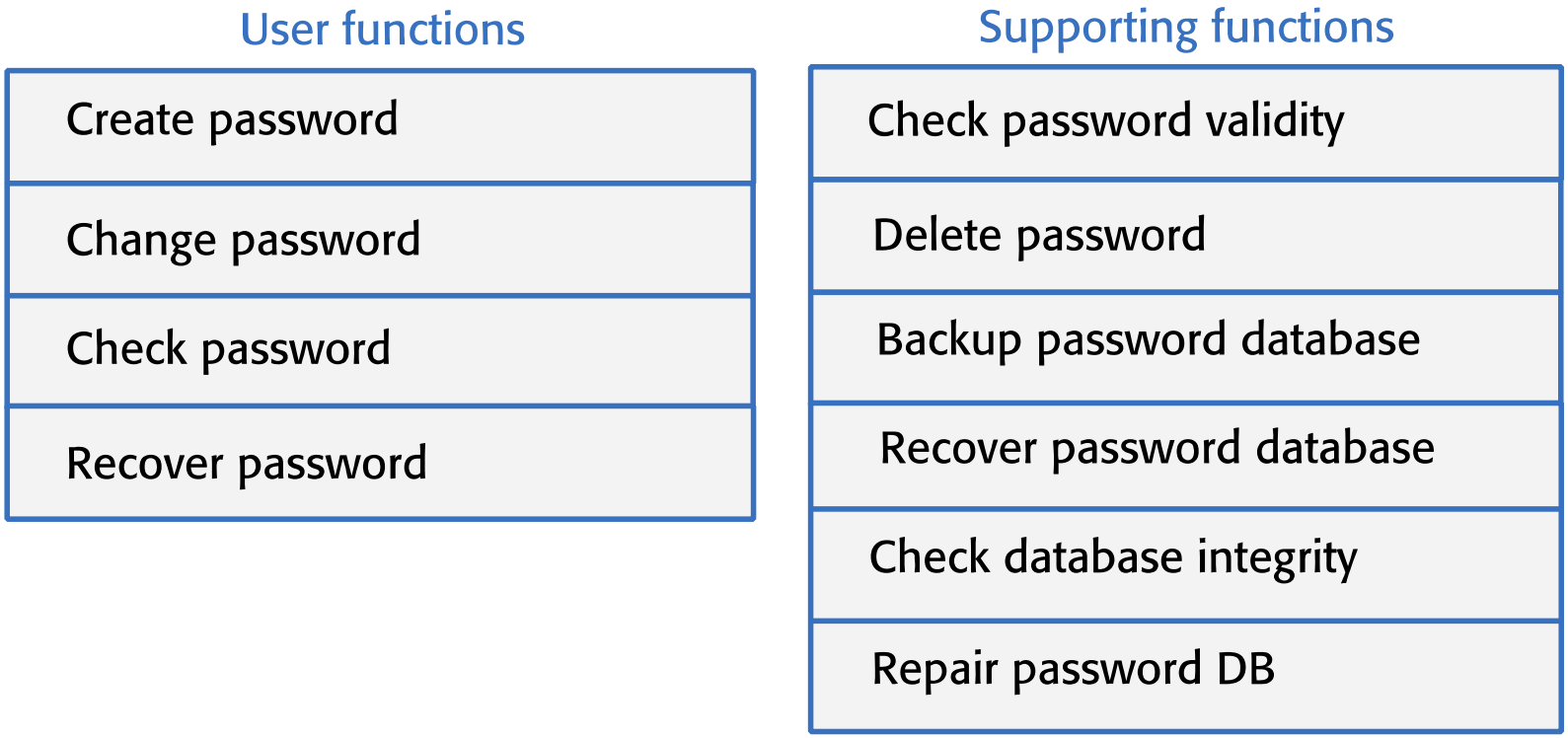
- ***Independently deployable***
  Each microservice runs in its **own process** and is **independently deployable**, using automated systems.

Microservice characteristics

- A well-designed microservice should have **high cohesion and low coupling**.
  - **Cohesion (internal)** is a measure of the number of relationships that parts of a component have with each other. High cohesion means that all of the parts that are needed to deliver the component's functionality are included in the component.
  - **Coupling (external)** is a measure of the number of relationships that one component has with other components in the system. Low coupling means that components do not have many relationships with other components.

- Each microservice should have a **single responsibility** i.e. it should do one thing only and it should do it well.
  - However, 'one thing only' is difficult to define in a way that's applicable to all services.
  - Responsibility does not always mean a single, functional activity.

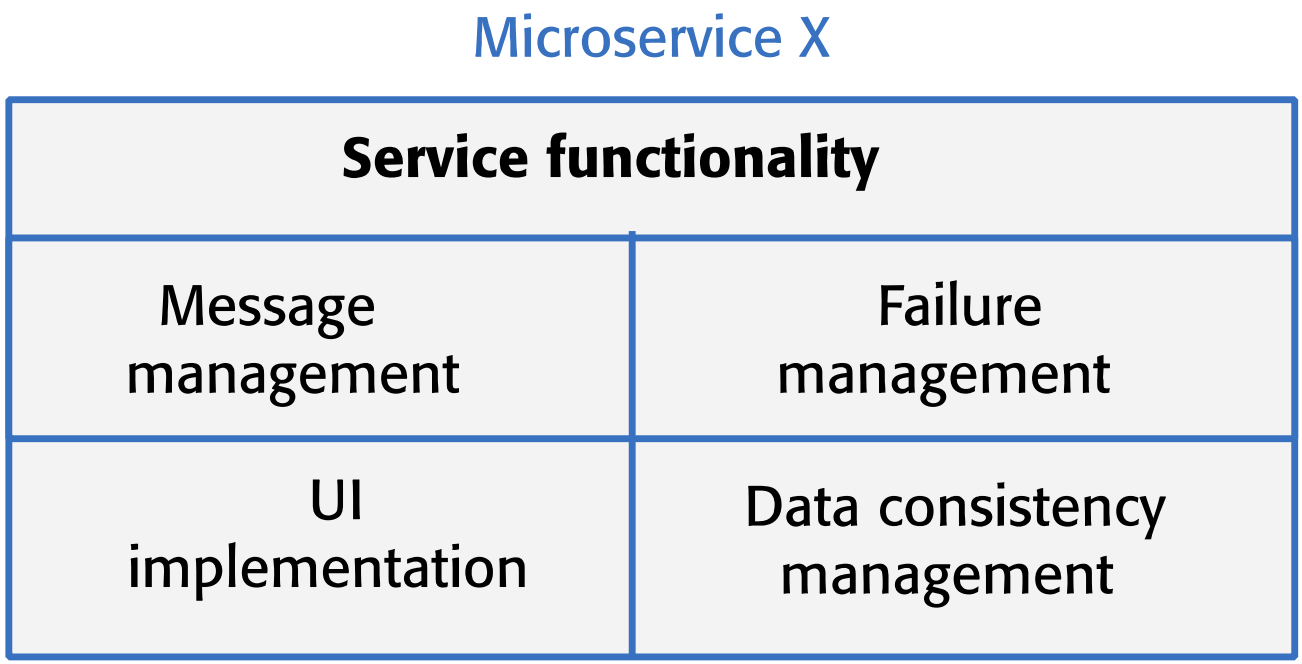
Password management service



- The range of functionality that might be included in a password management microservice

Microservice support code

- This code includes the set of tools, libraries, utilities, and helper functions
- **NOT** part of the core business logic of a microservice
- **BUT** to support the microservice's operation and integration within a larger microservices architecture.



Microservices architecture

- A microservices architecture is an *architectural style* – a tried and tested way using small, independent services that each perform a specific function.
- This architectural style addresses two problems:
  - **Independent Updates**: Unlike **monolithic** applications, where any change requires rebuilding, retesting, and redeploying the entire system, microservices allow individual components to be updated independently without affecting others. This speeds up development and reduces the risk of unintended side effects.
  - **Targeted Scaling**: In a **monolithic** system, increasing demand requires scaling the entire application, which can be inefficient. Microservices enable **scaling only the parts of the system that need it**, optimizing resource use and improving performance for high-demand features.

Benefits of microservices architecture

- Microservices are **self-contained** and run in separate processes.
- In cloud-based systems, each microservice may be **deployed in its own container**. This means a microservice can be stopped and restarted without affecting other parts of the system.
- If the demand on a service increases, **service replicas** can be quickly created and deployed. These do not require a more powerful server so ‘scaling-out’ is, typically, much cheaper than ’scaling up’.

A photo printing system for mobile devices

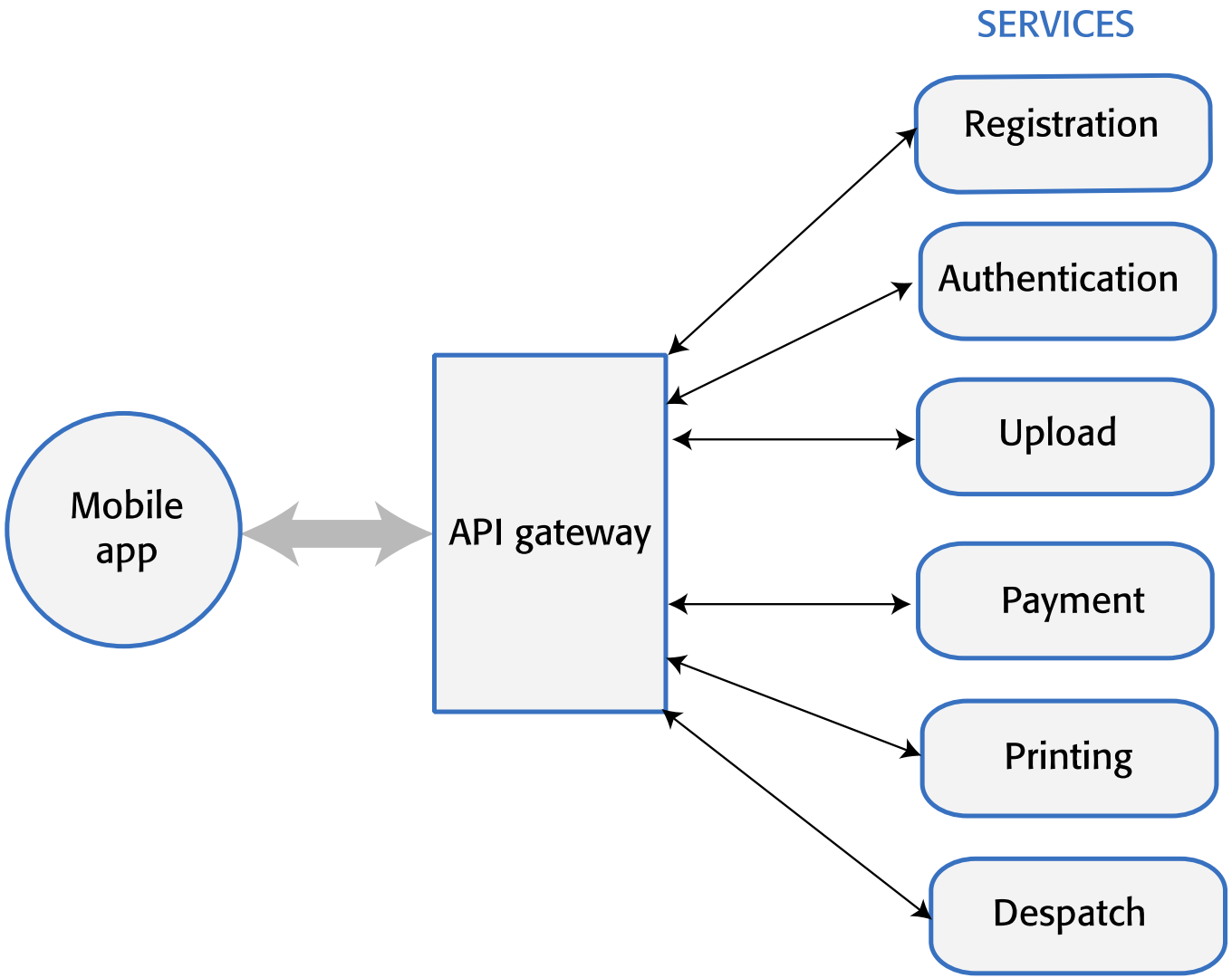
Imagine that you are developing a photo printing service for mobile devices.

- Users can upload photos to your server from their phone or specify photos from their Instagram account that they would like to be printed.
- Prints can be made at different sizes and on different media.
- Users can chose print size and print medium. For example, they may decide to print a picture onto a mug or a T-shirt.
- The prints or other media are prepared and then posted to their home.
- They pay for prints either using a payment service such as Android or Apple Pay or by registering a credit card with the printing service provider.

A monolithic client-server example

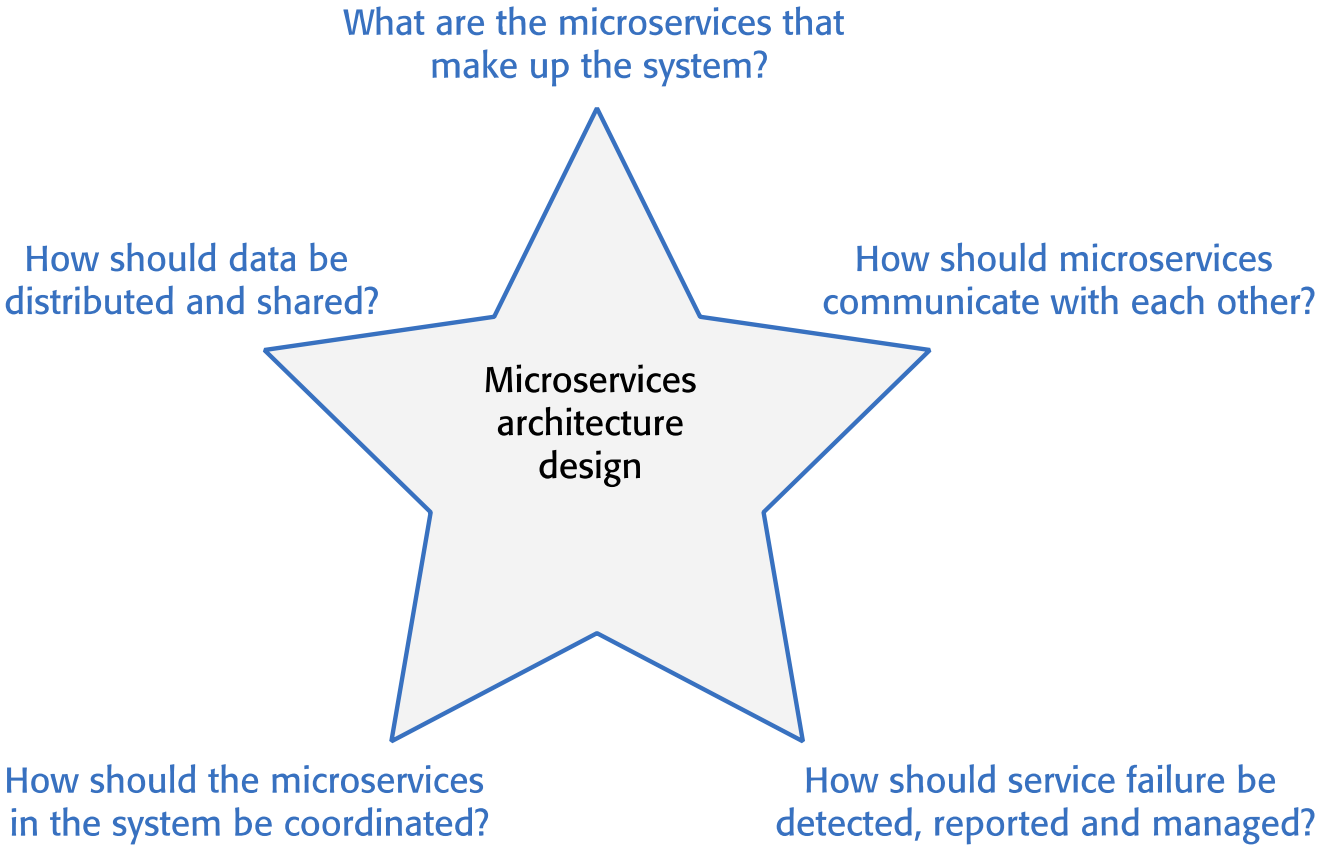
- All functionalities (photo management, order processing, payment, shipping, etc.) are implemented in a single codebase. This means any change to one part of the system (e.g., updating the payment processing logic) requires deploying the entire application again.
- All data (user, photos, orders, payments) is stored in a common, centralized database.

A microservices architecture for a photo printing system



Microservices architecture - key design questions

- Each service decides how to best provide its functionality, so technology choices should be made by the service implementation team, not the system architect.
- Although individual microservices are independent, they have to **coordinate and communicate** to provide the overall system service.



Microservice communication

- Microservices communicate by **exchanging messages**.
- A message that is sent between services includes **some administrative information**, a **request** and **the data** required to deliver the requested service.
- Services return a **response** to service request messages.
  - An authentication service may send a message to a login service that includes the name input by the user.
  - The response may be a token associated with a valid user name or might be an error saying that there is no registered user.

Decomposition guidelines

- **Balance fine-grain functionality and system performance**
  - Single-function services => many microservices but require communication between services to perform user functions. This slows the system due to the overhead of packaging and unpackaging messages.
- **Follow the 'common closure principle'**
  - Elements of a system that are likely to be **changed at the same time** should be **located within the same service**. Most new and changed requirements should therefore only affect a single service.
- **Associate services with business capabilities**
  - This means mapping out the various microservices that align with each business capability, ensuring each service is focused on delivering the required functionality.
- **Design services so that they only have access to the data that they need**
  - If there is an overlap between the data used by different services, you need a mechanism to propagate data changes to all services using the same data.

Service communications

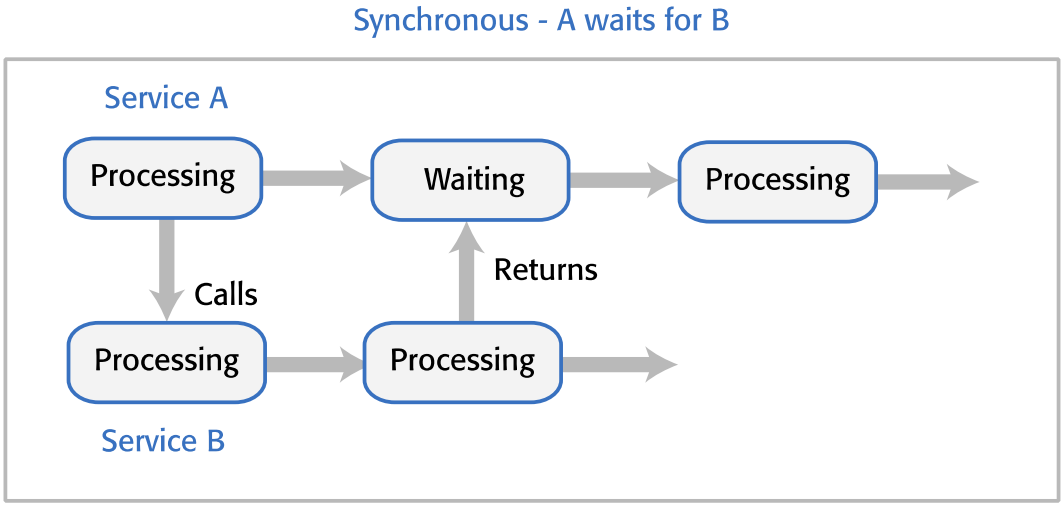
- Services communicate by exchanging messages that include information about the **originator** of the message, as well as the data that is the **input** to or **output** from the request.

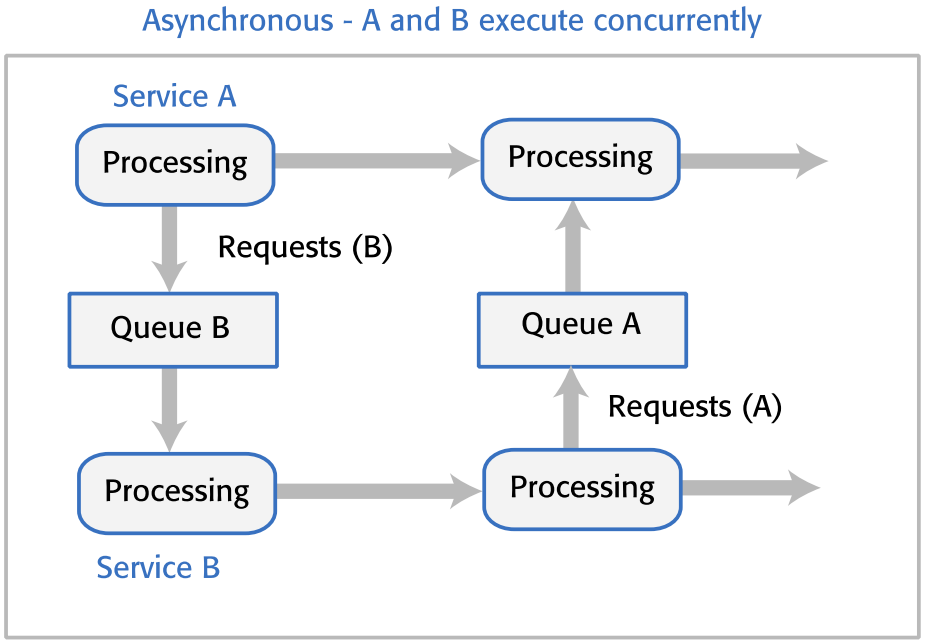
- When you are designing a microservices architecture, you have to establish a standard for communications that all microservices should follow. Some of the key decisions that you have to make are
  - should service interaction be **synchronous** or **asynchronous**?
  - should services **communicate directly** or via **message broker** middleware?
  - what **protocol** should be used for messages exchanged between services?

Synchronous and asynchronous interaction

- In a **synchronous** interaction, service A issues a request to service B. Service A then suspends processing while B is processing the request.
  - It **waits until** service B has returned the required information before continuing execution.
- In an **asynchronous** interaction, service A issues the request that is queued for processing by service B. A then **continues processing without waiting** for B to finish its computations.
  - Sometime later, service B completes the earlier request from service A and queues the result to be retrieved by A.
  - Service A, therefore, has to **check its queue periodically** to see if a result is available.

Synchronous and asynchronous microservice interaction





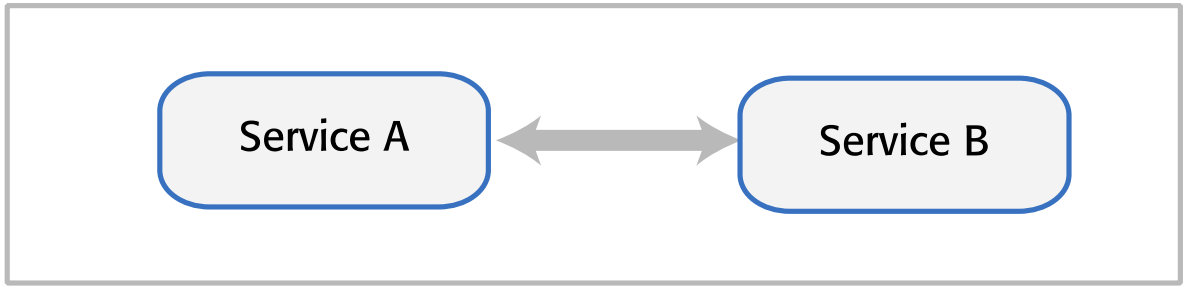
Direct and indirect service communication

- **Direct** service communication requires that interacting services **know each other's address**.
  - The services interact by sending requests directly to these addresses.

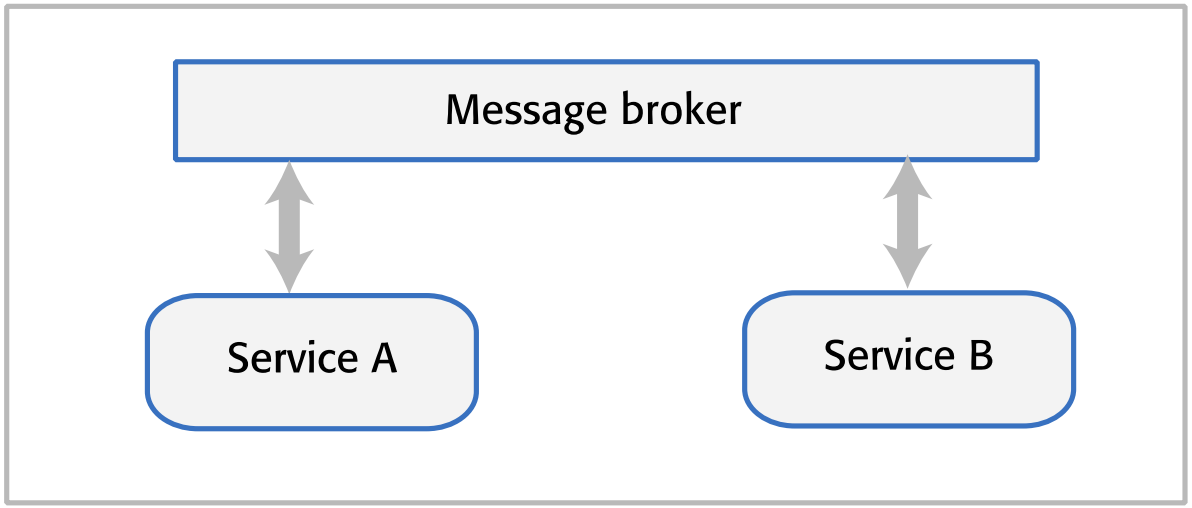
- **Indirect** communication involves naming the service that is required and sending that request to **a message broker** middleware (sometimes called a message bus).
  - The message broker is then responsible for **finding the service** that can fulfil the service request.

Direct and indirect service communication

Direct communication - A and B send messages to each other



Indirect communication - A and B communicate through a message broker



Direct and indirect service communication

- **Direct Service Communication:**
  - **Faster and simpler but requires knowing the URI of the requested service.**
  - **Failure if the URI changes.**
- **Indirect Communication:**
  - **Facilitates service versioning and can route requests to the most recent version.**
  - **It is complicated.**
- **Message Brokers:**
  - **Route requests and may handle message translation.**
  - **With Message Protocols (define message structure and data requirements between services)**
  - **Support synchronous and asynchronous interactions.**
  - **Simplify service modifications and replacements but increase system complexity.**

Microservice data design

- You should **isolate data within each service** with as little data sharing as possible.
- If data sharing is unavoidable, you should design microservices so that **most sharing** is '**read-only**', with a minimal number of services responsible for data **updates**.
- If services are replicated in your system, you must include a mechanism that can keep the database copies used by replica services consistent.

Inconsistency management

- **Concurrent updates to shared data** have the potential to cause **database inconsistency**.
- An ACID (atomicity, consistency, isolation, and durability) transaction is impractical in a microservices architecture.
  - A transaction is all-or-nothing, and a transaction brings the system from one valid state to another, maintaining data integrity.
  - In a microservices architecture, transactions may span multiple services, each with its own database.
  - Systems that use microservices have to be designed to tolerate some degree of data inconsistency.

Inconsistency management

- **Two types of inconsistency have to be managed:**
  - **Dependent data inconsistency**
    - The actions or failures of **one service** can cause the data managed by **another service** to become inconsistent.
    - A user places an online order for a book, triggering below services:
      - Stock Management Service: Reduces the number of books in stock by 1 and increases the number of books "pending sale" by 1.
      - Order Service: Places the order in a queue of orders to be fulfilled.
      - The stock level for the book becomes incorrect if the order service fails.
    - To address this:
      - **Service Failure Detection:** Implement mechanisms to detect when a service failure occurs.
      - **Compensating Transaction:** If the order service fails, initiate a compensating transaction in the stock management service **to restore the stock level** by incrementing it for the unfulfilled order.

Inconsistency management

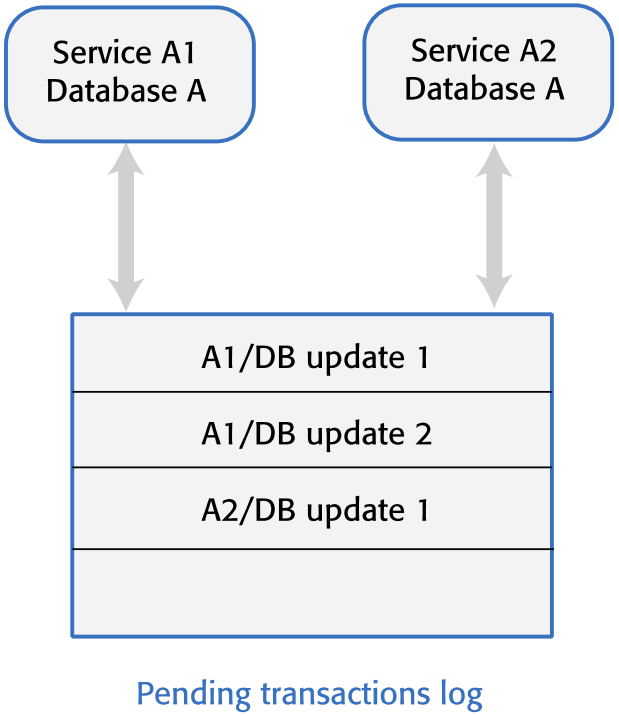
- **Two types of inconsistency have to be managed:**
  - **Replica inconsistency**
    - There are several replicas of the same service, with their own database copy and each updates its own copy of the service data.
    - Two identical instances of the stock management service (Service A and Service B) each with their own stock database.
      - Service A updates the stock for book X.
      - Service B updates the stock for book Y.
      - Problem: Discrepancies can arise if these updates are not synchronized, leading to inconsistent stock data across replicas.
    - You need a way of making these databases '**eventually consistent**' so that all replicas are working on the same data eventually.

Eventual consistency

- Eventual consistency is a situation where the system guarantees that the databases will *eventually become consistent*.
- You can implement eventual consistency by **maintaining a transaction log (also related to Service analytics)**.

Eventual consistency

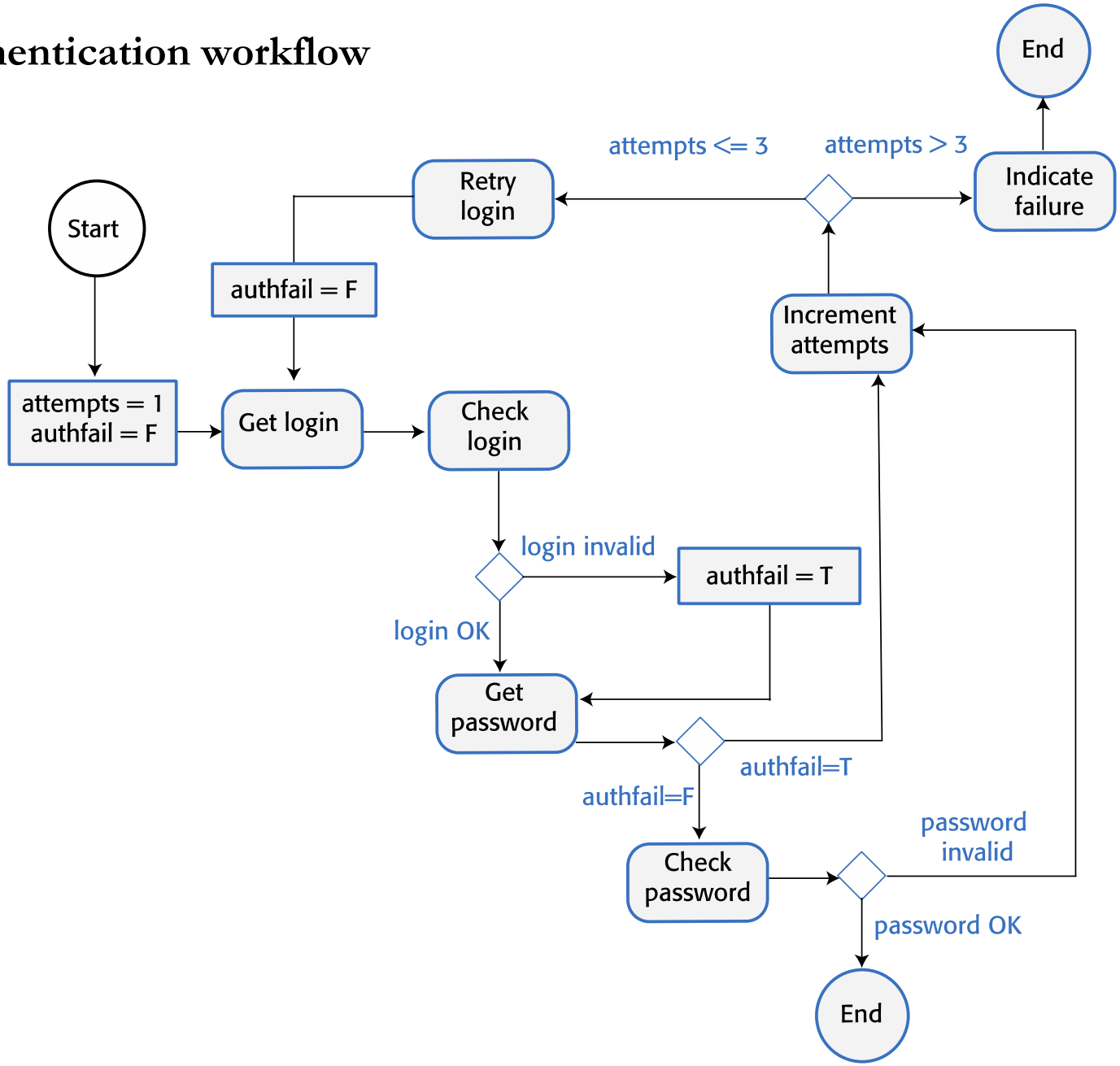
- When a database change is made, this is recorded on a '**pending updates**' log.
- Other service instances look at this log, update their own database and indicate that they have made the change.



Service coordination

- Most user sessions involve a **series of interactions** in which operations have to be carried out in a **specific order**.
- This is called a **workflow**.
  - An authentication workflow for UID/password authentication shows the steps involved in authenticating a user.
    - Define Workflow Steps
    - Central Coordinator to ensure the correct order
    - State Management
    - Error Handling and Recovery (retry mechanisms and compensating actions)
    - Service Communication

Authentication workflow



Failure types in a microservices system

- ***Internal service failure***
  These are conditions that are **detected by the service** and can be **reported to the service client** in an error message. An example of this type of failure is a service that takes a URL as an input and discovers that this is an invalid link.

- ***External service failure***
  These failures have an **external cause**, which affects the availability of a service. Failure may cause the service to become unresponsive and actions have to be taken to **restart the service**.

- ***Service performance failure***
  The performance of the service degrades **to an unacceptable level**. This may be due to a heavy load or an internal problem with the service. External service monitoring can be used to **detect performance failures** and **unresponsive services**.

Timeouts and circuit breakers

- How to detect failure
  - A **timeout is a counter** that this associated with the service requests and starts running when the request is made.
  - Once the counter reaches some predefined value, such as 10 seconds, the calling service **assumes that the service request has failed** and acts accordingly.

- The problem with the timeout approach is that every service call to a 'failed service' is delayed by the timeout value so the whole system slows down.
  - Instead of using timeouts explicitly when a service call is made, another option is using a **circuit breaker**. Like an electrical circuit breaker, this immediately denies access to a failed service without the delays associated with timeouts.

Using a circuit breaker to cope with service failure