Chapter 10
Microservices Architecture

Slides prepared by Ian Sommerville, modified by Ali El Hajj

Slides 6, 7, 8, 11, 12, 13, and 14 are based on: The Cloud Computing Book: The Future of Computing Explained, By Douglas E. Comer, 1st Edition, 2021

Software services

- A software service is accessed through its published interface and all details of the service implementation are hidden.
- Services do not maintain any internal state. State information is either stored in a database or is maintained by the service requestor.
- As there is no local state, services can be dynamically reallocated from one virtual server to another and replicated across several servers.
- 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.

Modern Web Services

- The idea of 'big' Web Services emerged in the early 2000s.
- These were based on XML-based protocols and standards such as SOAP (Simple object Access protocol) for service interaction and WSDL for interface description.
- It takes a significant amount of time to analyze the XML messages and extract the encoded data, which slows down systems built using these web services.
- These services usually shared a database and provided an API that was used by the system's user interface module. In practice, it was not easy to scale or move individual services without affecting other parts of the system.
- Most software services don't need the generality that's inherent in the design of web service protocols.
- Consequently, modern service-oriented systems, use simpler, 'lighter weight' service-interaction protocols that have lower overheads and, consequently, faster execution.
- RESTful services involve a lower overhead and are used by many organizations implementing service-based systems.

RESTful web services

- REST (REpresentational State Transfer) is an architectural style based on transferring representations of resources from a server to a client.
  - It is a web based standard architecture and uses HTTP Protocol
  - It is simpler than SOAP/WSDL for implementing web services.
  - RESTful services involve a lower overhead and are used in implementing service-based systems.
  - The fundamental element in a RESTful architecture is a resource: A resource is a data element such as a catalog, a medical record, or a document.
  - Resources have multiple representations and formats: WORD, PDF, etc.
- A REST API (also known as RESTful API) is an application programming interface that conforms to REST architectural style and allows for interaction with RESTful web services.
  - In this architecture, every component is a resource identified by URIs/ global IDs.
  - REST API uses various representation to represent a resource like text, JSON, XML but JSON is the most popular one.

REST vs SOAP

- When a client request is made via a RESTful API, a representation of the state of the resource is transferred to the requestor via HTTP in one of several formats : JSON, HTML, XML, XLT, Python, PHP, or plain text.
  - REST doesn't pose any restriction for a specific format in representing resources, it all depends on the choice and requirement of the project and developer.
- Many legacy systems may still follow SOAP, while REST came later and is faster in web-based systems.
- REST is a set of guidelines with flexible implementation. SOAP has specific requirements such as XML messaging.
- REST APIs are lightweight, making them more suitable to Internet of Things (IoT), mobile applications, etc.
- SOAP web services have built-in security and transaction compliance which makes them heavier.

Monolithic Applications in a Data Center

- *A monolithic* application is constructed as a single, self-contained piece of software:
  - Large and complex.
  - Contains the code needed to handle all steps
  - Organized into a main program plus a set of *functions*
- A tenant can launch and use a VM to run monolithic applications. Disadvantages:
  - monolithic applications cannot be replicated as quickly as cloud-native applications.
  - starting a VM has higher overhead than starting a container.
  - all code must be downloaded when the application starts, even if pieces are not used.

Problems with monolithic applications

- The whole system has to be rebuilt, re-tested and re-deployed when any change is made.
  - This can be a slow process as changes to one part of the system can affect other components.
  - Frequent application updates are therefore impossible.
- As the demand on the system increases, the whole system has to be scaled, even if the demand is localized to a small number of components.
  - Larger servers must be used increasing cost.
  - Starting up a larger server can take several minutes.

The Microservices Approach

- An alternative way to structure software that takes advantage of multiple computers in a data center:
  - Divide each application into pieces, allowing the pieces to scale independently, as needed
- A microservices architecture is a tried and tested *architectural style* that addresses the problems with monolithic applications.
- *Microservices architecture* divides functionality into multiple, independent applications. Each of the applications:
  - Much smaller than a monolithic program,
  - Handles only one function.
  - Applications communicate over a network.
- The microservices approach can be used in two ways:
  - Implement a new application.
  - *Disaggregation D*ivide an existing monolithic application

Figure 12.1 Illustration of a monolithic application with all the functions needed to support online shopping built into a single program.

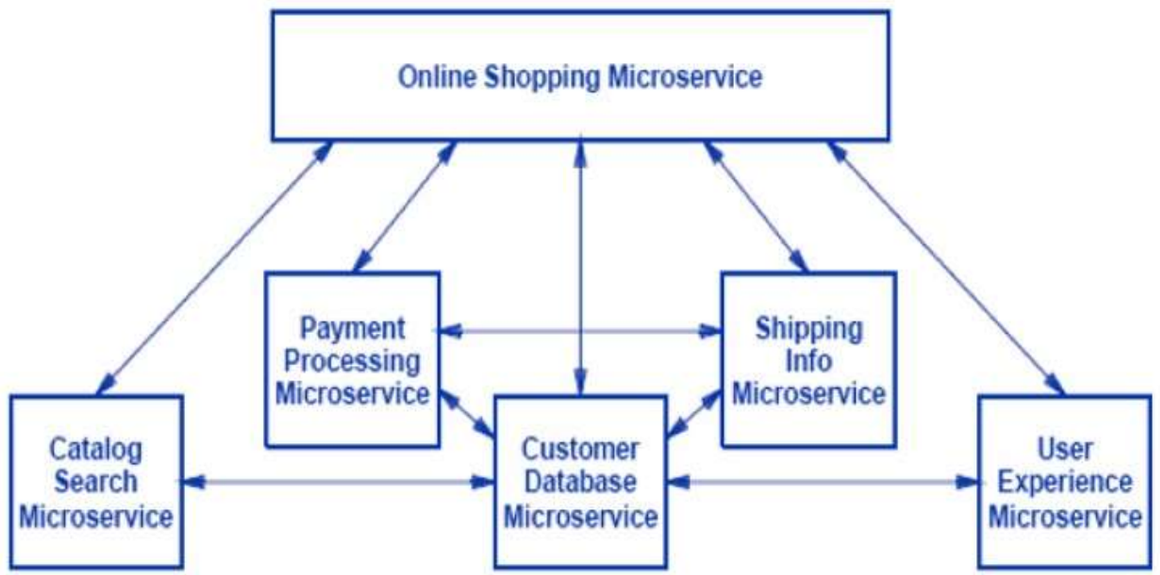


Figure 12.2 Illustration of the shopping application disaggregated into a set of microservices that communicate with one another.

Microservices

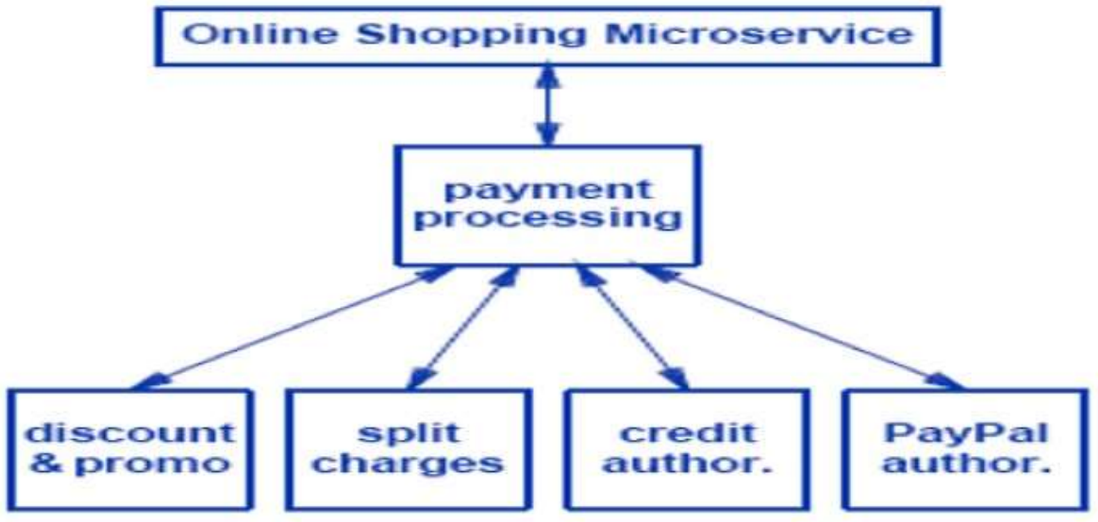
- Microservices are small-scale, stateless, services that have a single responsibility or business function. They are combined to create applications.
- Instead of relying on a shared database and other services, microservices are completely independent with their own database and UI management code.
- Replacing or replicating a microservice should therefore be possible without having to change any other services in the system.
- It is recommended to design Cloud-based software products that are adaptable, scaleable and resilient, around a microservices architecture.

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, 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.
- ***Business-oriented***
  Microservices should implement business capabilities and needs, rather than simply provide a technical service.

Microservices Granularity

- How much functionality should be packed into each microservice? How much should be a microservice size?
- Example payment processing: Should a separate microservice be used for each function or should they all be collected into a single microservice?
  - Separate microservices have advantages, such as the ability to scale independently.
  - Adding microservices introduces more management complexity and increases both communication and the security attack surface



The Advantages Of Microservices

- Microservices introduces the extra overhead of running multiple, small applications and using network communication among the pieces instead of internal function invocation.
- In a cloud environment, microservices advantages outweigh the overhead:
  - Advantages for software development
  - Advantages for operations and maintenance.

Advantages For Software Development

- Smaller scope and better modularity: Software engineers focus on one small piece of the problem at a time and define clean interfaces. They are less likely to make mistakes
- Smaller teams: The resulting code will be more uniform and less prone to errors.
- Less complexity: The monolithic approach creates complexity, and complexity leads to errors.
  - Microservices approach eliminates global variables, and requires designers to document the exact interfaces among pieces
- Choice of programming language: software engineers can choose the best language for each service, instead of a single programming language`
- More extensive testing: Testing a monolithic program poses a challenge because the pieces can interact. Many combinations of inputs must be used.
  - With the microservices approach, each service can be tested independently.

Advantages For Operations And Maintenance

- **Rapid deployment**: Because each service is small, a given microservice can be created, tested, and deployed rapidly.
- **Independent upgrade of each service**: a new microservice version can be introduced without stopping existing applications or disturbing other microservices
- **Improved fault isolation**: When a problem occurs, the misbehaving microservice can be identifed and tested, while other microservices continue normal operations.
- **Better control of scaling**: Each microservice can be scaled independently to handle the load without scaling other services that are not heavily used.
- Compatibility with containers and orchestration systems: Because it is small and only performs one task, a microservice fits best into the container paradigm.
  - Microservices can be monitored, scaled, and load balanced by a container orchestration system, such as Kubernetes.

Disadvantages Of Microservices

- **Cascading errors:** one microservice can invoke another, which can invoke another, and so on. If one of the microservices fails, the failure may affect many others as well as the applications that use them.
- **Duplication of functionality and overlap**: When functionality is needed that differs slightly from an existing microservice, it is often easier to create a new one than to modify the existing microservice
- **Management complexity**: Microservices make management complex. Although orchestration and automation tools help, with hundreds of microservices running simultaneously, it can be difficult to understand their behaviors, interdependencies, and interactions.
- **Replication of data and transmission overhead**: Each microservice requires a copy of the needed data which leads to overhead.
- **Increased security attack surface**: Microservices approach creates multiple points that an attacker can try.
- **Workforce training**: software engineers need new skills to create software for microservices, including running cost, communication cost, granularity, etc.

Service deployment

- After a system has been developed and delivered, it has to be deployed on servers, monitored for problems and updated as new versions become available.
  - When a system is composed of tens or even hundreds of microservices, deployment of the system is more complex than for monolithic systems.

- The service development teams decide which programming language, database, libraries and other support software should be used to implement their service. Consequently, there is no 'standard' deployment configuration for all services.

- It is now normal practice for microservice development teams to be responsible for deployment and service management as well as software development and to use continuous deployment.

  - Continuous deployment means that as soon as a change to a service has been made and validated, the modified service is redeployed.

Deployment automation

- Continuous deployment depends on automation so that as soon as a change is committed, a series of automated activities is triggered to test the software.

- If the software 'passes' these tests, it then enters another automation pipeline that packages and deploys the software.

- The deployment of a new service version starts with the programmer committing the code changes to a code management system such as Git.

- This triggers a set of automated tests that run using the modified service. If all service tests run successfully, a new version of the system that incorporates the changed service is created.

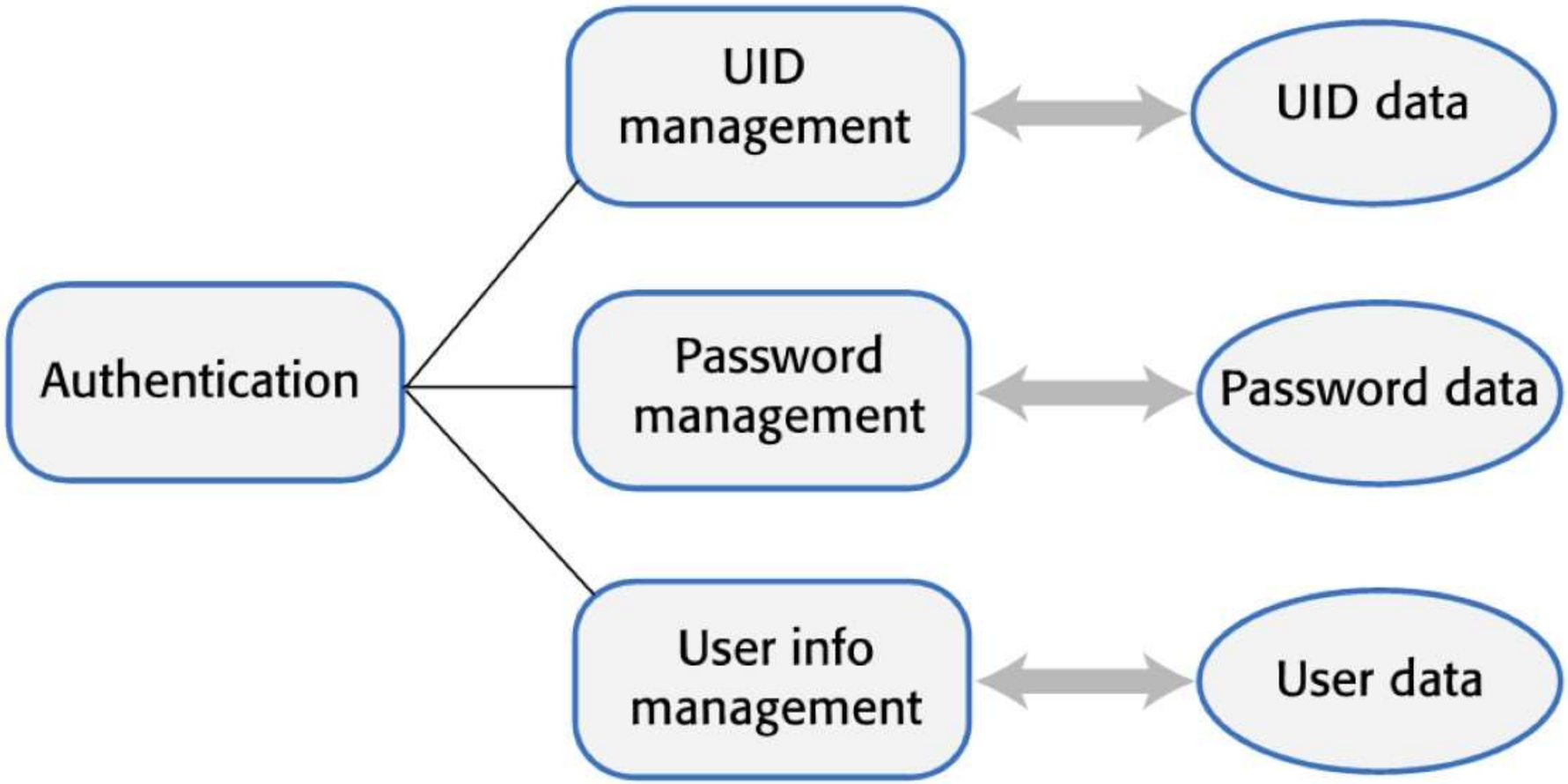
- Another set of automated system tests are then executed. If these run successfully, the service is ready for deployment.

A microservice example

- System authentication
  - 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. To identify the microservices that might be used in the authentication system, you need to break down the coarse-grain features into more detailed functions.

Functional breakdown of authentication features

Authentication microservices



Microservice communication

- Microservices communicate by exchanging messages.
- A message that is sent between services includes some administrative information, a service 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.

Microservice design characteristics

- A well-designed microservice should have high cohesion and low coupling.
  - Cohesion 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 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.
  - Example: Password management microservice

Password management microservice functionality

User functions
Create password
Change password
Check password
Recover password

Supporting functions
Check password validity
Delete password
Backup password database
Recover password database
Check database integrity
Repair password DB

Microservice support code

Independence of microservices means that each service has to include support code that may be shared in a mono-lithic system. Support code that is needed in all microservices includes:

- Message management is responsible for processing (parsing, formatting…) incoming and outgoing messages.
- Failure management is needed when the microservice cannot complete a requested operation or when it returns an error or does not reply.
- Data consistency management is needed when the data used in a microservice are also used by other services.
- For complete independence, each microservice should maintain its own tailored user interface

Microservice X

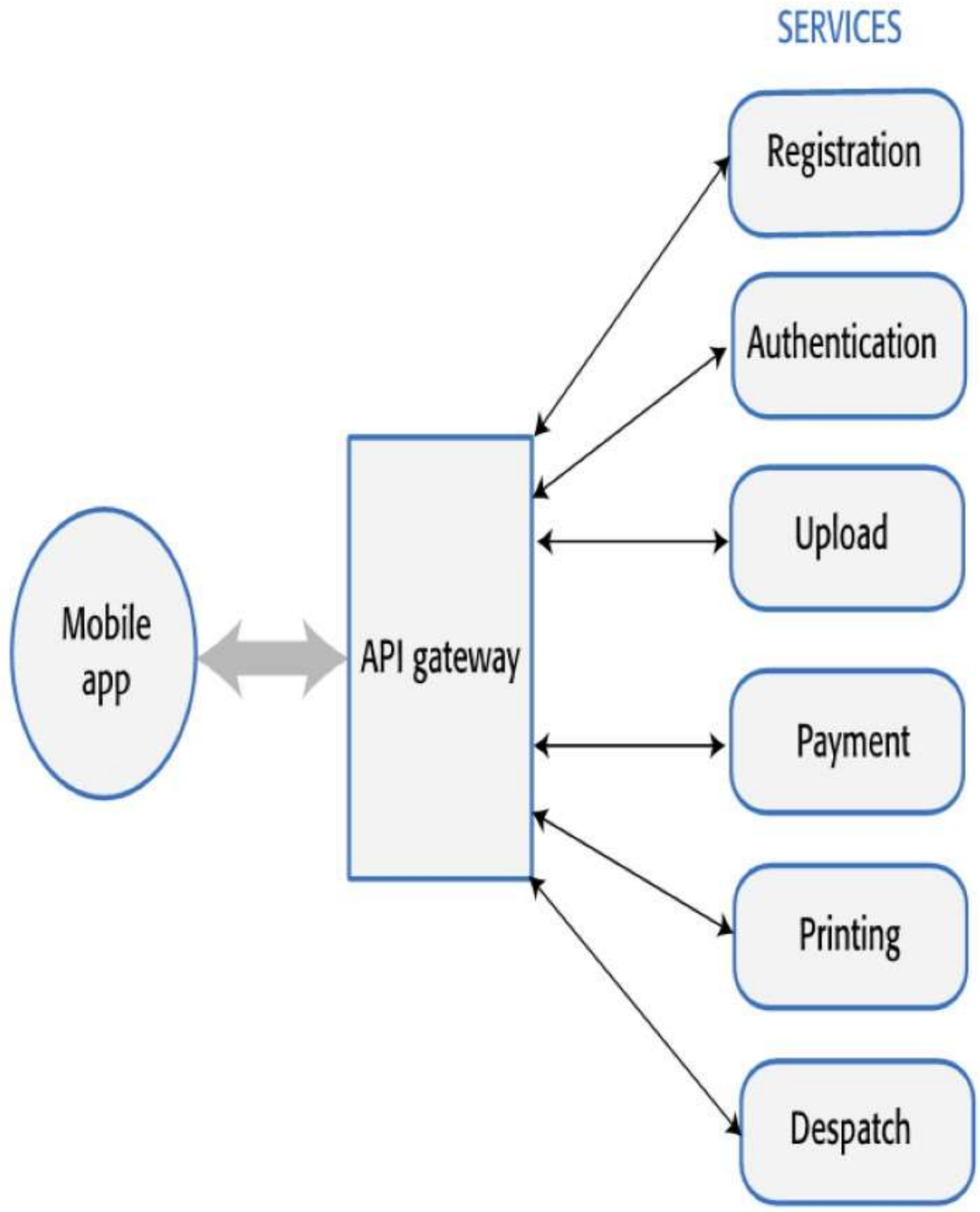
Service functionality	
Message management	Failure management
UI implementation	Data consistency management

A photo printing service for mobile devices

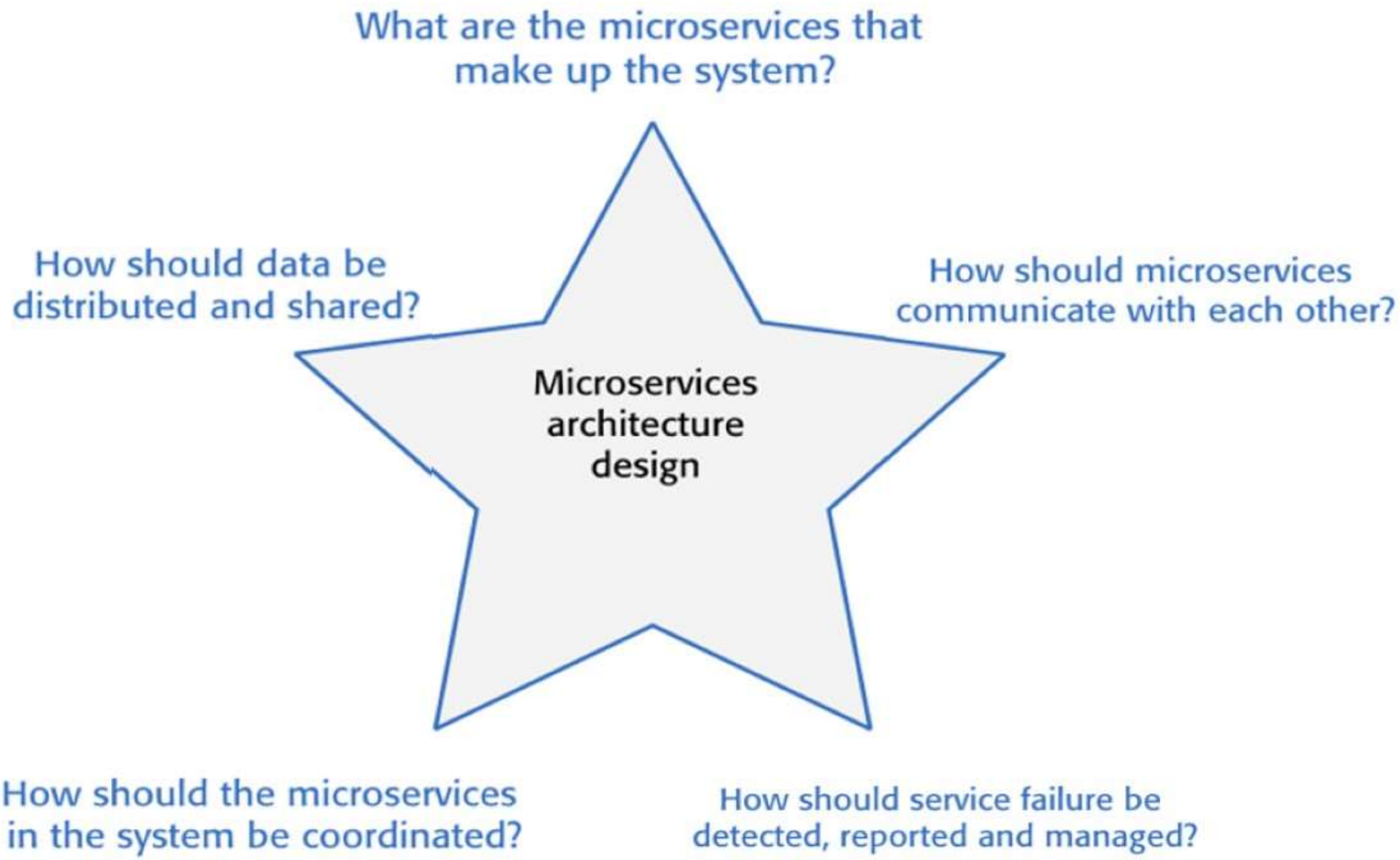
- In a photo printing service for mobile devices, users can upload photos to server from their phone or specify photos from their Instagram account that they would like to be printed.
- Users can chose print size and print medium. For example, they may decide to print a picture onto a mug or a T-shirt. The prints are prepared and then posted to their home. They pay for prints using a payment service or by registering a credit card with the printing service provider.

A microservices architecture for photo printing

- Each microservice may be deployed in its own container. It can be stopped and restarted without affecting other parts.
- If the demand on a service increases, replicas can be quickly created and deployed with no need to a powerful server.
- The API gateway insulates the user app from the system's microservices. It is a single point of contact and translates requests from the app into calls to the microservices.
  - The app does not need to know what communication protocol is being used.
  - It is possible to change the service decomposition by splitting or combining services without affecting the client app.



Microservices Architecture Design Issues



Decomposition guidelines

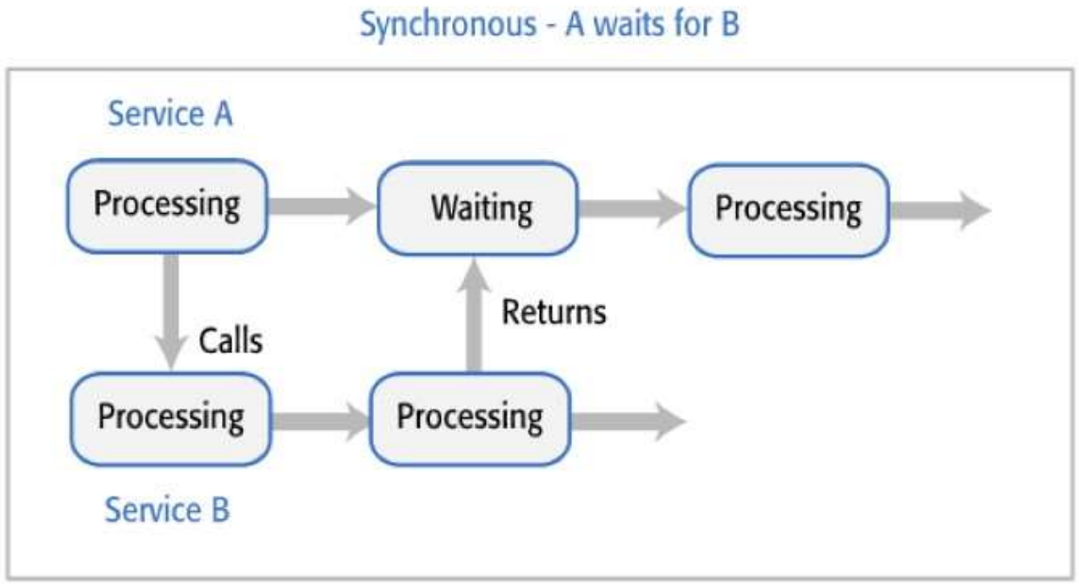
- Too many microservices mean that service communications slows down the system.
- Too few microservices means that each service will be larger, so changing them is likely to be more difficult.
- General design guidelines to decompose a system into microservice:
  - Balance fine-grain functionality and system performance
    - Single-function services require service communications to implement user functionality. This slows down a system because of the need for each service to bundle and unbundle messages sent from other services.
  - 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
    - A business capability is a business functionality that is the responsibility of an individual or a group. You should identify the services that are required to support each business capability.
  - 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.
- One possible starting point for microservice identification is to look at the data that services have to manage.

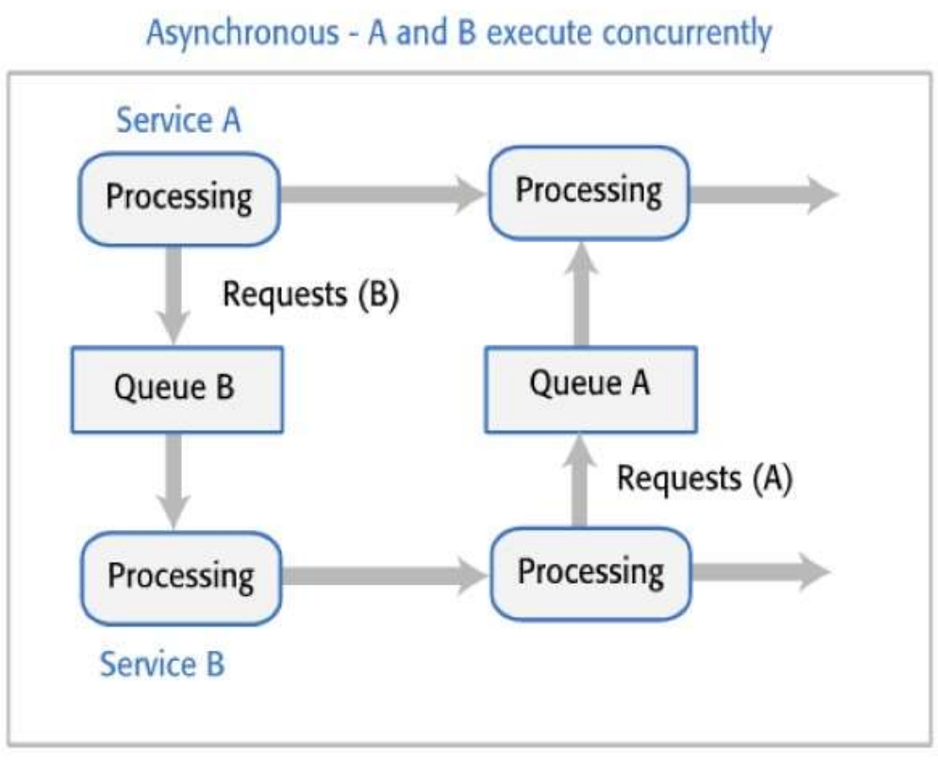
Service communications

- 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 and waits until service B has processed the request and returned the required information.
- In an asynchronous interaction, service A issues the request to service B and continues processing. Later, service B completes the 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 programs are easier to write and understand, but asynchronous interaction is often more efficient. Services that interact asynchronously are loosely coupled.
- Start with synchronous interaction, then rewrite some services to interact asynchronously if you find that the performance is not good enough





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. Direct service communication is simple, easy to understand, and faster to develop products.
- Indirect communication involves naming the service that is required and sending that request to a message broker (sometimes called a message bus): The message broker (such as API gateway, RabbitMQ) is then responsible for finding the service that can fulfil the request.
- Message brokers can support synchronous and asynchronous interactions. It is easier to modify and replace services without affecting the clients. However, the overall system becomes more complex.

Direct communication - A and B send messages to each other

Indirect communication - A and B communicate through a message broker

Microservice data design

Microservices are interacting systems rather than individual units. This means:

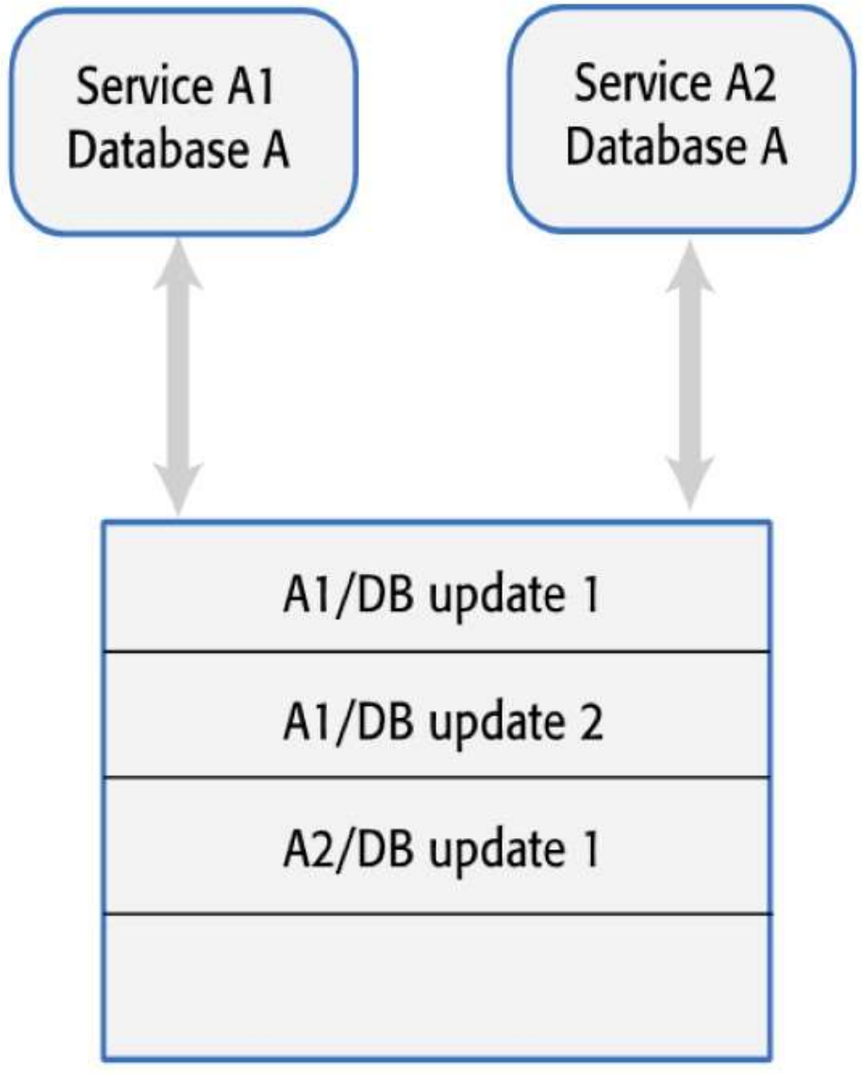
- You should isolate data within each system service with as little data sharing as possible.
- If data sharing is unavoidable, you should design microservices so that most sharing is ‘read-only’, with a minimal number of services responsible for data updates.
- If services are replicated in your system,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

Inconsistency management

- An ACID transaction bundles a set of data updates into a single unit so that either all updates are completed or none of them are. ACID transactions are impractical in a microservices architecture unless you can confine the data involved in the transaction to a single microservice.
- The databases used by different microservices need not be completely consistent all of the time, and some degree of data inconsistency need to be tolerated. 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.
  - Replica inconsistency: There are several replicas of the same service that are executing concurrently. These all have their own database copy and each updates its own copy of the service data. You need a way of making these databases 'eventually consistent' so that all replicas are working on the same data.

Eventual consistency

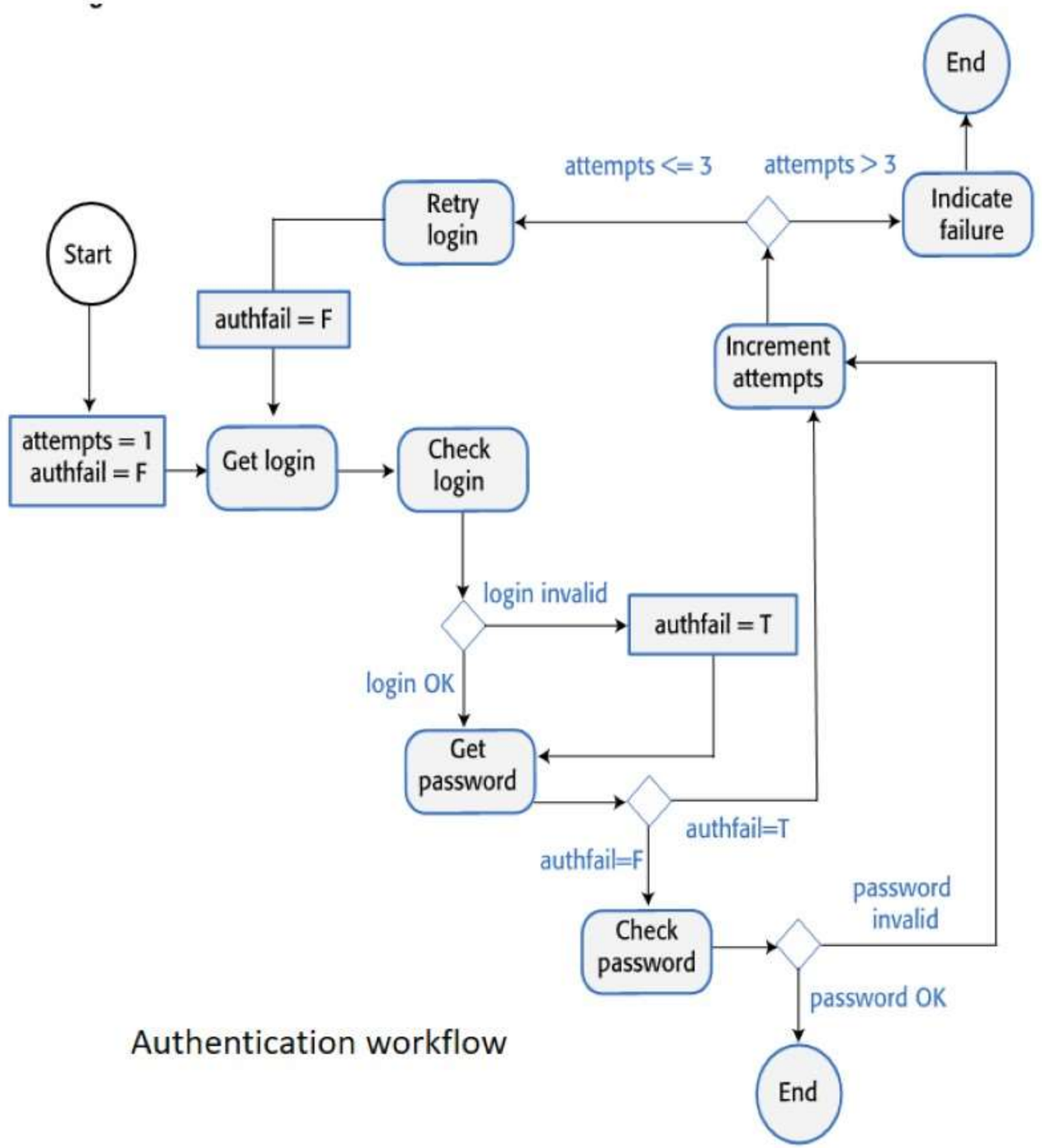
- Using eventual consistency, the system guarantees that the databases will eventually become consistent.
- This can be implemented by maintaining a transaction log.
- When a database change is made, this is recorded on a 'pending updates' log.
- Other service instances look at this log, update their own database and indicate that they have made the update.
- After all services have updated their own database, the transaction is removed from the log.



Pending transactions log

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.
  - In this example, the user is allowed 3 login attempts before the system indicates that the login has failed.

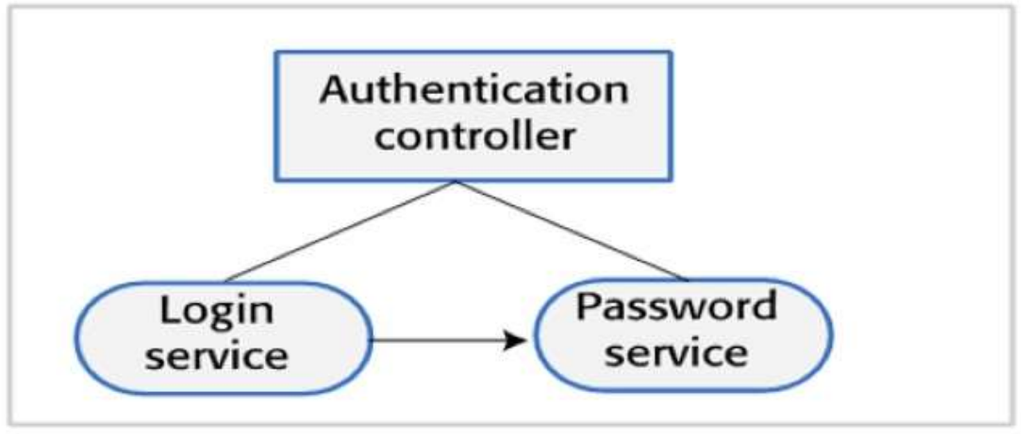


Authentication workflow

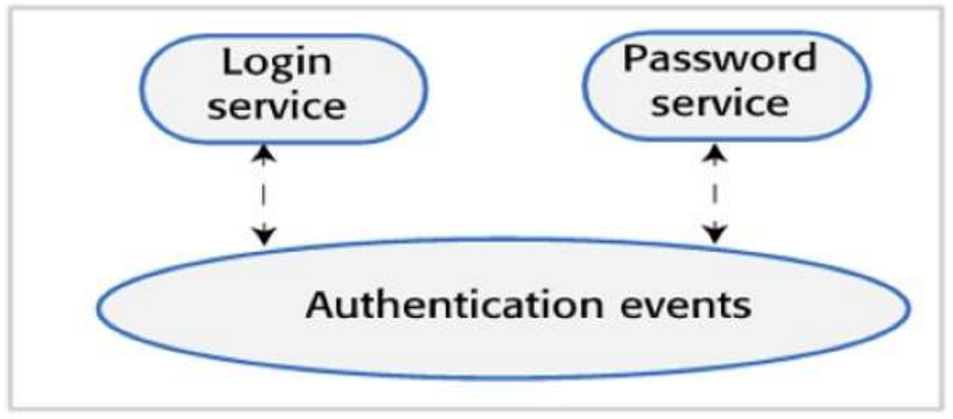
Orchestration and choreography

- One way to implement this workflow is to define the workflow explicitly (either in a workflow language or in code) and to have a separate service that executes the workflow by calling the component services in turn. This is called “orchestration,”
- In an orchestrated approach, if a service fails, the controller knows which service has failed and where the failure has occurred
- An alternative approach is called “choreography” in which each service emits an event to indicate that it has completed its processing. Other services watch for events and react accordingly. An additional software such as a message broker supports a publish and subscribe mechanism. Services “publish” events to other services and “subscribe” to those events that they need. Unlike orchestration, microservices works in parallel. The entire system work on event-based architecture, where a service collects data from a message bus and perform the business logic and in return submit data to another message bus.
- Choreographed workflows are harder to debug. If a failure occurs, it is not immediately obvious what service has failed.

Service orchestration



Service choreography



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.

The simplest way to report microservice failures is to use HTTP status codes, which indicate whether or not a request has succeeded.

Timeouts and circuit breakers

- 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, use circuit breaker that immediately denies access to a failed service without the delays associated with timeouts.