ESD

Enterprise Applications & Business Processes

Enterprise Solution

Typically, an **enterprise solution** refers to one or more **enterprise applications** that **produce, process and exchange data** in order to **fulfil business requirements**

- E.g., automate the book order handling & shipping process in a bookstore; aggregate all relevant information on a single interactive dashboard

3 Categories of Enterprise Applications

1. **Commercial-Off-The-Shelf**
    - Buy from an external vendor. The software package is pre-built with "best practices". Not adapted to specific requirements of a customer
    - Pros: Provides required functionality out of the box; may be more reliable with tech supports; If using standardized technologies or providing documented APIs, easier to exchange data with other applications
    - Cons: If using proprietary technologies without APIs or customizability, may be more difficult to be used for a process; may be expensive to buy
2. **Custom**
    - Built from scratch either using in-house or external developers according to specific requirements of a customer.
    - Pros: Internally built may make it easier to use it for the processes within the same enterprise and exchange data with other applications, but may be harder to work with applications from other enterprises
    - Cons: May be costly and unreliable in building the functionality; may be expensive to build from scratch

3. **Legacy**
    - Old systems that have been built many years before using old technologies that may not be widely used now
    - Pros: Provides required functionality; may be more stable
    - Cons: Interface, data format, etc. may be old and incompatible with new technologies, making it harder to exchange data with newer applications; may be expensive to maintain for a long term

Monolithic Applications vs Microservices

A **microservice** may be defined as **a single unit** that implements only **one or a few (instead of many) functionality** needed to support business requirements and **can be used** by other applications (or microservices) **over the network** in **a standard interface** that is **independent of programming languages and platforms**

Characteristics of Microservices

- Microservices are "**loosely coupled**" with each other
- It can be implemented in a programming **language of its own**;
- It can be deployed and run on a **platform of its own**; E.g., one microservice runs on Windows; the other on Linux;
- It usually has its **own data store** when a data store is needed; E.g., an Order microservice has its own database with an Order table; a Customer microservice has its own database with a Customer table
- It can be **scaled independently** E.g., instances of the same microservice that is more frequently used than others can be deployed to many machines to support concurrent processing
- Its implementation can be **changed independently**, as long as its interface for invocation remains the same
- Microservices **exchange data** with each other through a standard interface by using **commonly used data formats and communication technologies**

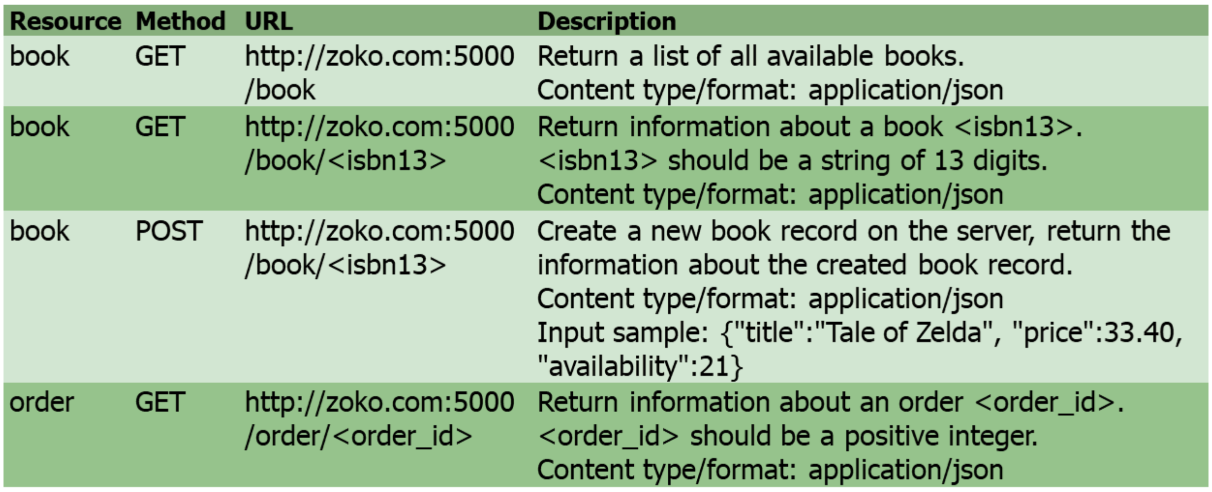
Rest APIs and JSON

Rest Operations

Uses standard HTTP operations (methods) explicitly

Operation	Meaning
POST	Create resource on the server
GET	Retrieve resource from the server
PUT	Update resource on the server
DELETE	Delete resource from the server

API Docs / Documentation of REST APIs:



Resource	Method	URL	Description
book	GET	http://zoko.com:5000 /book	Return a list of all available books. Content type/format: application/json
book	GET	http://zoko.com:5000 /book/<isbn13>	Return information about a book <isbn13>. <isbn13> should be a string of 13 digits. Content type/format: application/json
book	POST	http://zoko.com:5000 /book/<isbn13>	Create a new book record on the server, return the information about the created book record. Content type/format: application/json Input sample: {"title":"Tale of Zelda", "price":33.40, "availability":21}
order	GET	http://zoko.com:5000 /order/<order_id>	Return information about an order <order_id>. <order_id> should be a positive integer. Content type/format: application/json

REST Characteristics

- **Client-server** model
- **Simple uniform (standardized) interface**
- Each API is **language and platform agnostic** (implementation can be in any language of choice, deployment can be on any platform of choice)
- Typically supports only the **HTTP / HTTPS** transport
- Supports a **variety of data formats:** TEXT, JSON, HTML, XML, JPEG etc
- Each API is often **stateless:** The server processes each API invocation in isolation; it doesn't maintain states about the client or previous API invocations.

JSON Data Format

- **JSON** (**JavaScript Object Notation**) is a lightweight data-interchange format
- **Self-describing:** both human and machine readable
- Uses a text format independent of programming languages and platforms

```
{"books": [
   {"availability": 2,
     "isbn13": "9781129474251",
     "price": 21.5,
     "title": "SQL in Nutshell"
   },
   {"availability": 25,
     "isbn13": "9781349471231",
     "price": 99.4,
     "title": "Understanding People"
   },
   …
]}
```

Service and SOA Concepts

Types of (Micro)Services

Atomic (Micro)Service

- The **(micro)service** implements the expected functionalities by itself **independent of** other (micro)services or applications within an enterprise context (except for commonly used applications, such as Apache HTTP server, MySQL server)
- An **atomic microservice** is often simple, providing one or a few functionality related to **one kind of entity** of a business concern (e.g., Book, Customer, etc.); the operations provided by the microservice are often simple CRUD of the data entities.
- An **atomic service** may be complex, providing many functionalities involving multiple kinds of entities, which is **not** recommended by Microservices Architecture.

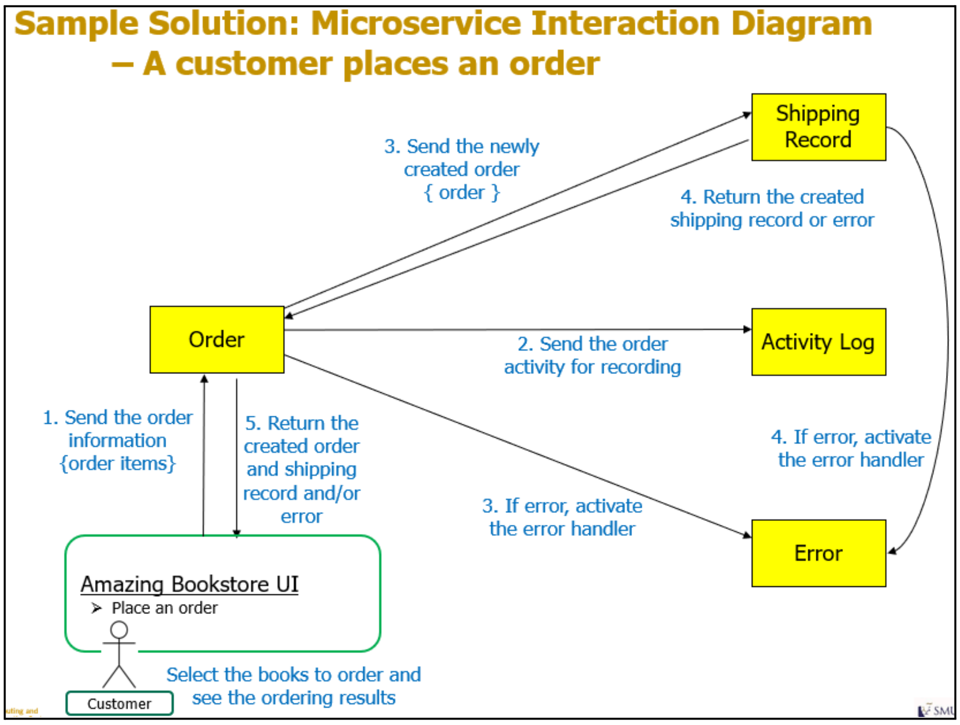
Wrapper Service

- It usually does not provide new functionalities; it mainly exposes some functionalities of another application (e.g., a legacy system, an external system) as a service.

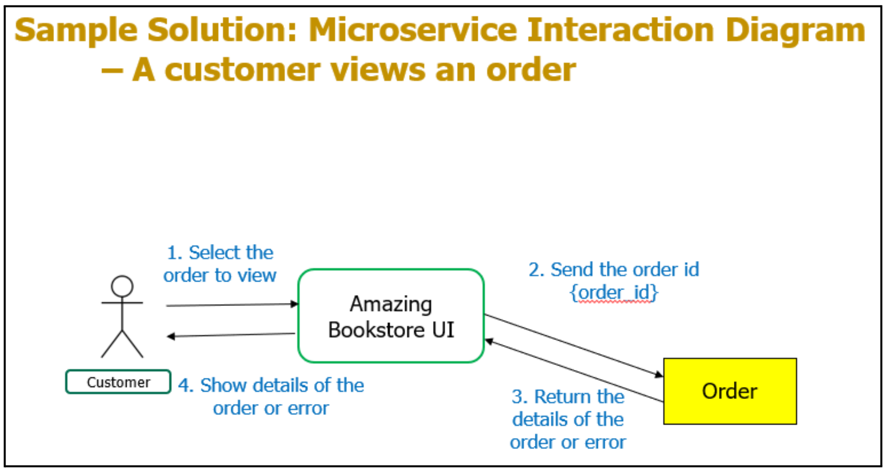
Composite (Micro)Service

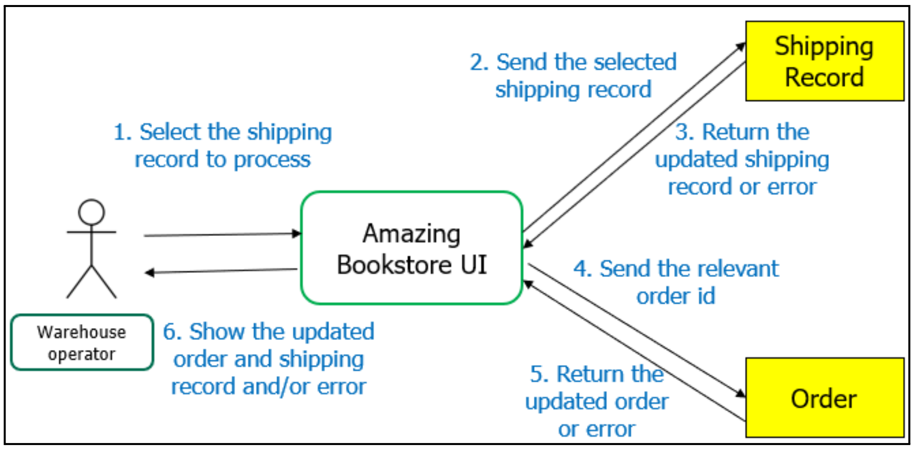
- "**Composite**" implies that the (micro)service implements combines a set/sequence of other services or applications together according to a business process (e.g., Place an order, Process a shipping record, Restock inventory)
- Often implemented via **inter-process communication (IPC) technologies**
- Often called **process (micro)service** as it is often used to implement a business process involving more than one kinds of data entities.
- **microservice** may use other (micro)services, but not a monolith directly

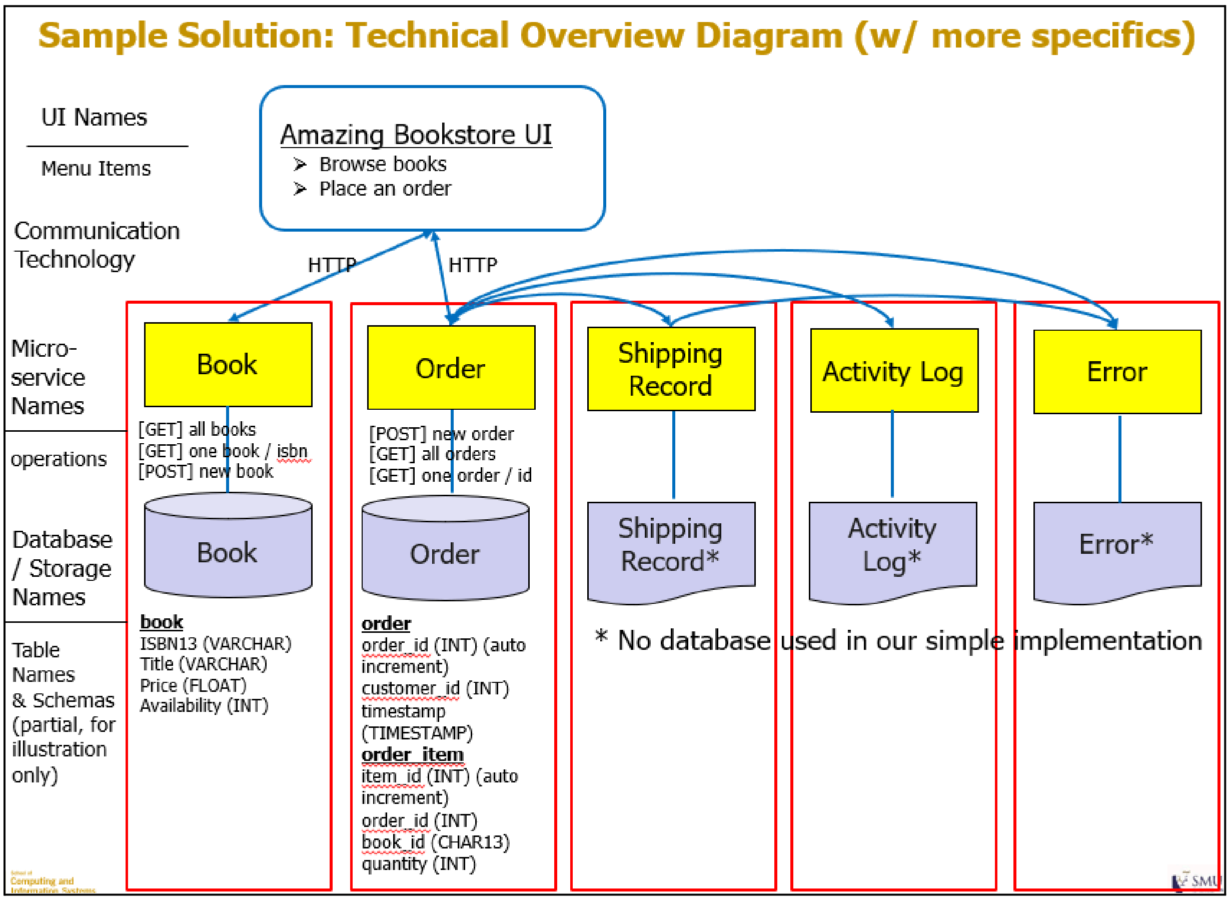
Microservices Design Exercise


Sample Solution: Microservice Interaction Diagram
– A customer places an order
Shipping Record
3. Send the newly created order { order }
4. Return the created shipping record or error
Order
Activity Log
2. Send the order activity for recording
1. Send the order information {order items}
5. Return the created order and shipping record and/or error
4. If error, activate the error handler
3. If error, activate the error handler
Error
Amazing Bookstore UI
Place an order
Customer
Select the books to order and see the ordering results

1. The Order microservice creates order record after receiving the order information from UI;
2. The Activity Log microservice gets notified about the order creation status, whether successfully or not;
3. If order creation succeeds, the order information is sent to create a shipping record; otherwise, the Error microservice gets notified;
4. The Shipping Record microservice returns its status (whether the shipping record is created successfully or not) back to the Order microservice; the Shipping Record microservice also informs the Error microservice if it encounters errors;
5. The Order microservice returns its status and the status of the shipping record back to UI.


Sample Solution: Microservice Interaction Diagram
– A customer views an order
1. Select the order to view
Amazing Bookstore UI
2. Send the order id {order_id}
Customer
4. Show details of the order or error
3. Return the details of the order or error
Order


Warehouse operator
1. Select the shipping record to process
Amazing Bookstore UI
2. Send the selected shipping record
Shipping Record
3. Return the updated shipping record or error
4. Send the relevant order id
Order
5. Return the updated order or error
6. Show the updated order and shipping record and/or error



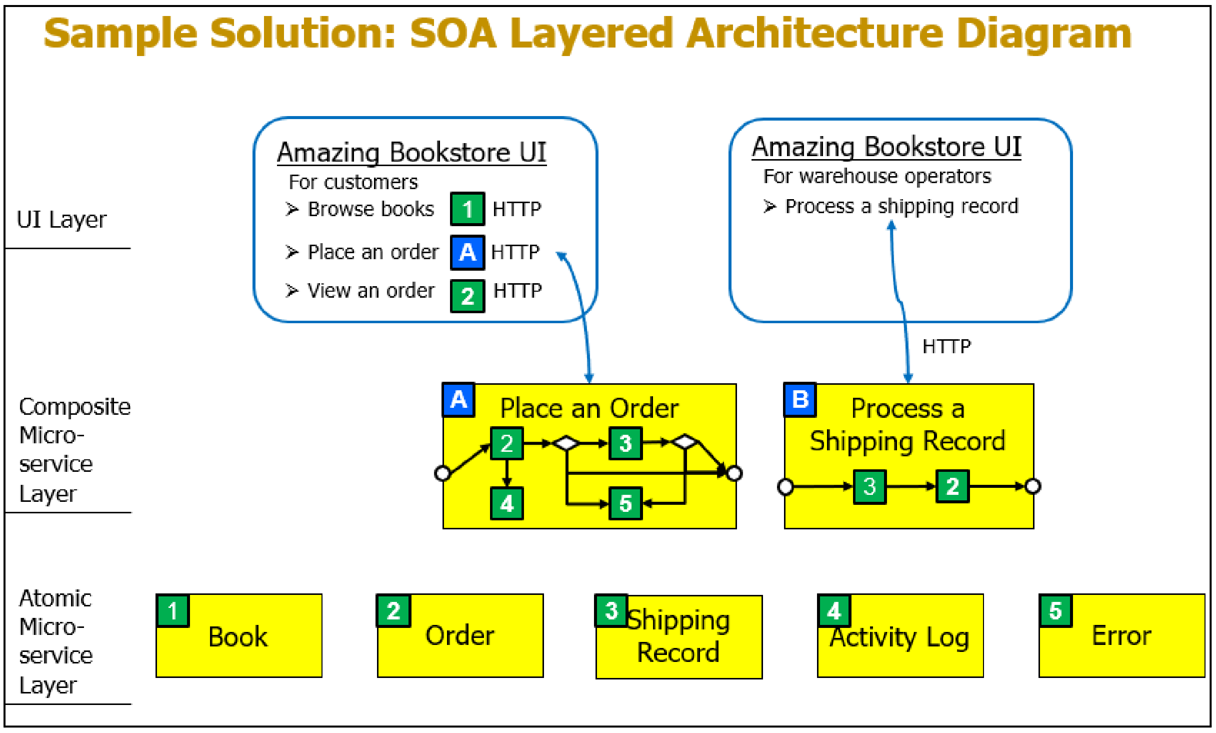
Pros and Cons of Microserve Design vs Monolithic Design

Pros: microservices are "loosely coupled" with each other

- Each microservice can be developed and deployed independently from each other.
- Each microservice usually has its own datastore, and can be scaled independently from each other.
- Each microservice can be changed independently, as long as its interface for invocation remains the same.

Cons: Development and Setup Overheads

- Each microservice needs code to handle data transport (e.g., JSON over HTTP) to enable interactions, less convenient than function/method calls directly supported in programming languages
- Interactions need to be implemented in a way that tolerate additional possible failures (e.g., network interruption, timeout, transaction violation, data consistency across microservices, etc.)
- Interactions among microservices may become complex, hard to understand, test, and maintain



SOA diagram has less details about the data stores for the microservices, but more details about the workflows within the composite microservices.

Choreographic vs Orchestrated Patterns

Choreographic architectural pattern

- No clear central controller of a business process
- Microservices can react to any data or event received without knowing who is the sender, and send out data without knowing who is the receiver
- Distributed coordination among microservices

Orchestrated architectural pattern

- A central controller, often a composite microservice, exists in a business process
- The controller interacts with other microservices to complete a business process
- Centralized coordination among microservices

API Docs

- Provide a reference to all the APIs available from all the (micro)services in an enterprise solution
- Define the **standard application programming interfaces** to the services, including the data formats used, communication technologies used, service names, etc.
- Every service consumer can look at the documents and know how to use the services, without the need of knowing the internal implementation of the services

Docker and Service Deployment

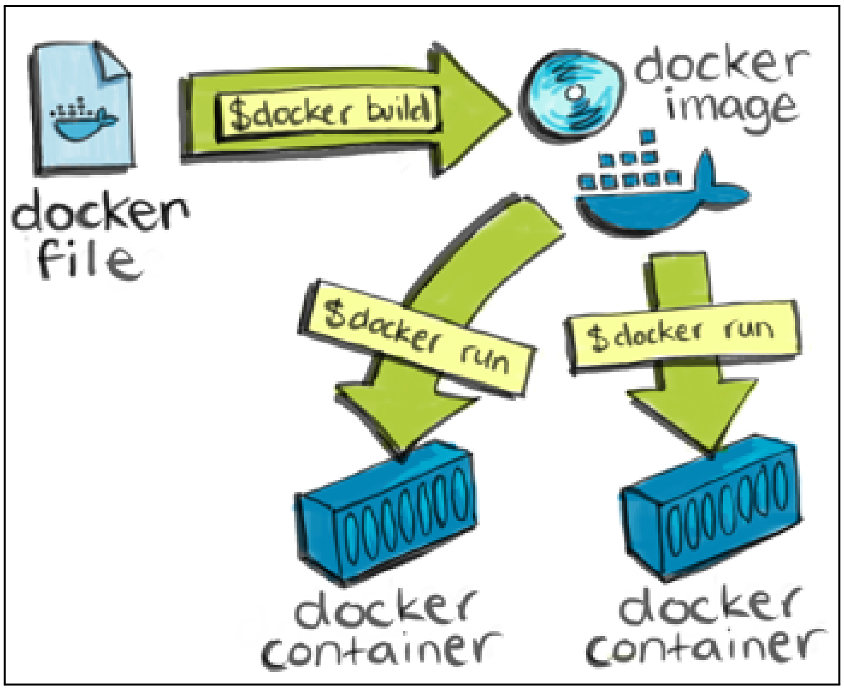
Containerization of a microservice

- Package necessary items needed to run a microservice into an **image** that can be transferred to and run as a **container** on another machine (think of the image as a disc)
    - Container: an instance of an image (i.e. running an instance of the disc)

Docker:

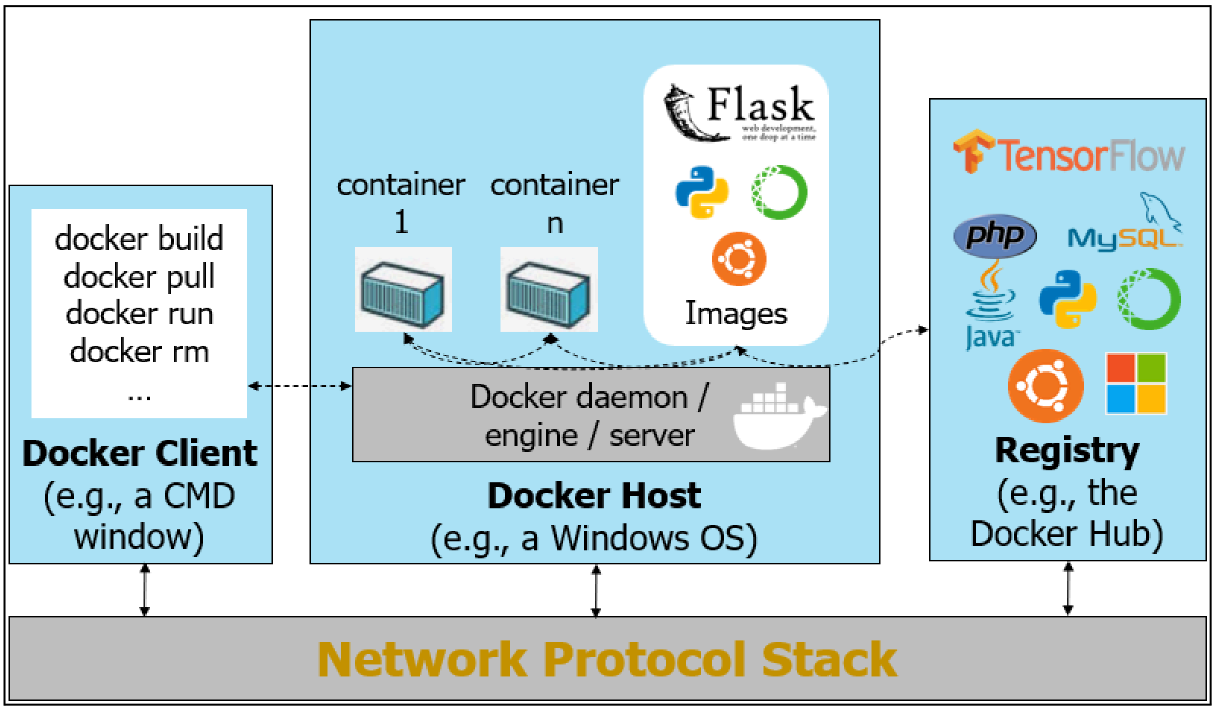
- an open **platform** for developers and system administrators to **package**, **transfer**, and **run** microservices (or any* application) on any* machine either on premise or on the cloud
- Docker is slightly slower to run than running directly, as there is extra software to run

Dockerfile contains the instructions for Docker to build an image (FROM, COPY, RUN, etc.)



A Docker container is a self-contained, runnable software application or service. On the other hand, a Docker image is the template loaded onto the container to run it, like a set of instructions.

Docker Architecture



Docker Host

The machine that installs Docker and runs the Docker engine (your computer)

Docker Client

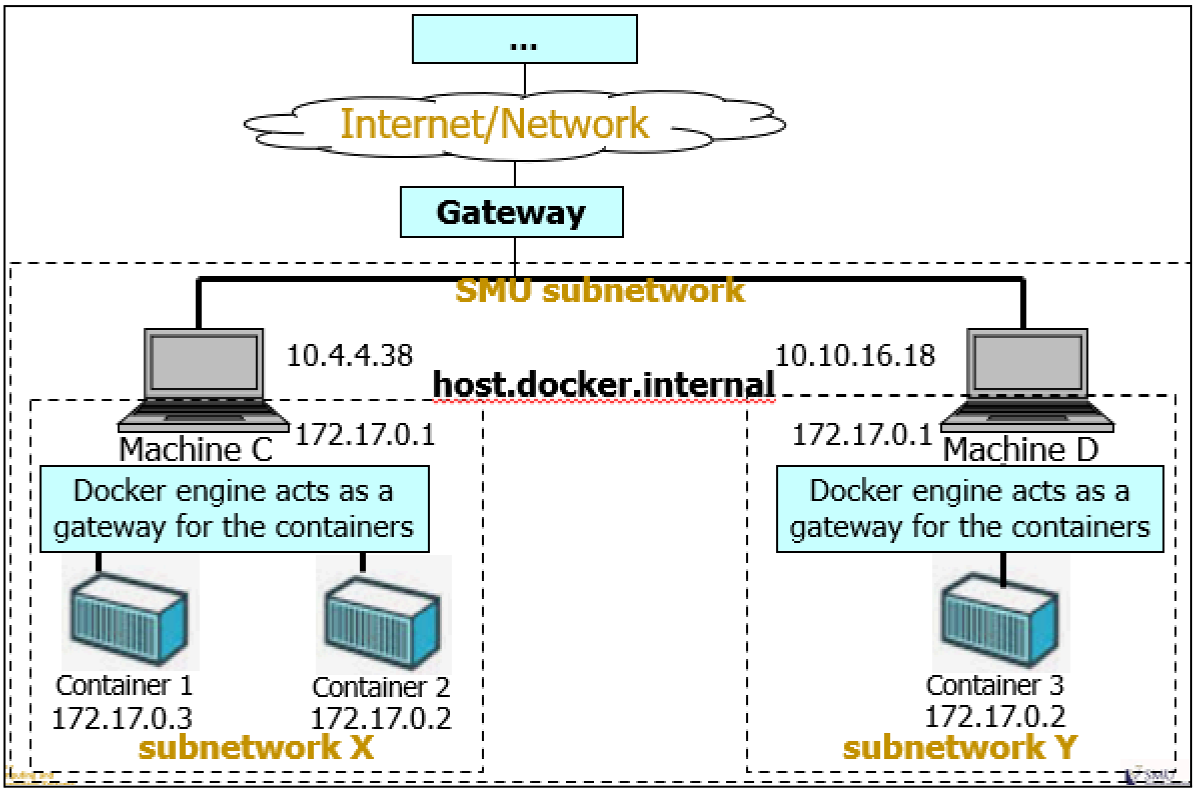
The command window that runs Docker commands. In the lab, it is the same machine, but the cmd can be run from other machines as well

- The docker commands (e.g. docker run, docker build) require <dockerid>, which is the id of the **docker host**

Registry

- Central storage location for distributing Docker images (uploading and downloading). The most common registry is Docker Hub, where developers can publish their images for others to use. Private registries can also be set up; however, Docker is configured to look for images on Docker Hub by default.
- When you use docker pull or docker run commands, Docker automatically retrieves the necessary images from the registry you have configured. On the other hand, when you use docker push, the image you specify gets uploaded to your configured registry.

Container Networking



The containers created by the Docker Engine (on your computer) will be in the same subnetwork. Different Docker Engines create different subnetworks.

By default, containers within the same subnetwork can communicate with each other.

For Docker containers to communicate across different subnetworks, they need **Network Address Translation (NAT)** through the Docker engine and Docker host, e.g., **port mapping**

- depends on many networking settings between the different Docker hosts.

An external client can directly access the host IPs in the external network but not the container IPs (because **internal IPs are invalid externally**)

- A common solution is to have an intermediary to **translate** between external and internal IPs
- Most of the time, the Docker engine (on your computer) is the intermediary

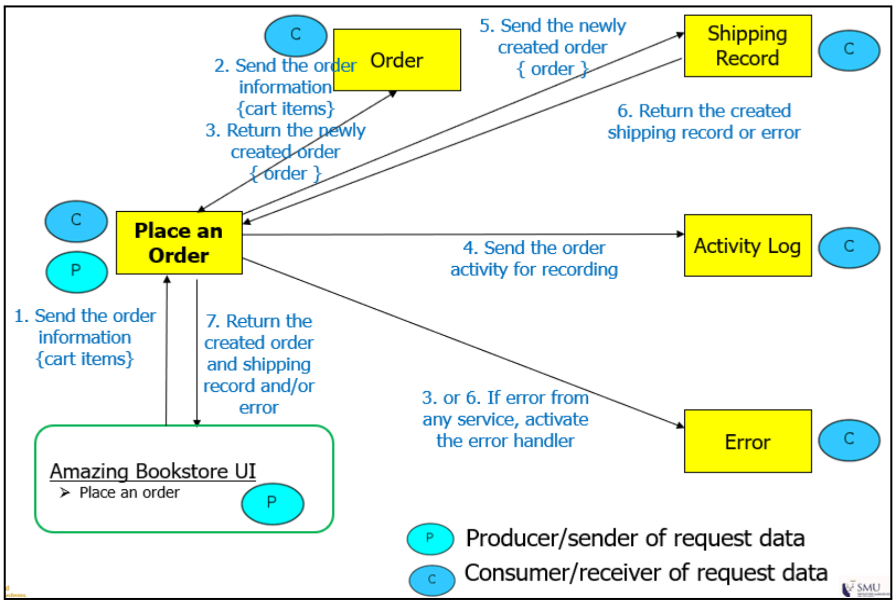
Network Address Translation

Port Mapping: redirects communication data from an IP:port number to another IP:port number

- In this case, it redirects data from the port number of your computer to the port number of the Docker container
  - The docker engine maps an (external) port of the docker host (a.k.a. host port) to an (internal) port of the container (a.k.a. container port); after that, all incoming request data going to the host port is forwarded to the container port of a particular container; the data replied from the container is forwarded by the docker engine and host back to the original client.

E.g. docker run -p 5100:5000 → local host port 5100 is mapped to container port 5000

Docker Compose


Order
2. Send the order information {cart items}
3. Return the newly created order { order }
5. Send the newly created order { order }
Shipping Record
6. Return the created shipping record or error
Place an Order
4. Send the order activity for recording
Activity Log
1. Send the order information {cart items}
7. Return the created order and shipping record and/or error
3. or 6. If error from any service, activate the error handler
Error
Amazing Bookstore UI
Place an order
C
P
Producer/sender of request data
Consumer/receiver of request data
SMU

In order to build and start running all of containers for the microservices,

- how many command prompt windows we need to open, and how many docker and python commands (e.g., “docker build”, “docker run”, "python book.py") we need to run?
- how many commands (e.g., “docker stop”, “docker rm”) we need to run to turn off/remove all the microservices?

A better automation is needed to manage the builds/runs/executions of the containers for microservices.

Docker Compose

- A tool for **defining** and **running** multi-containers
- Use **YAML** files to define the configurations of containers
- Use **`docker-compose`** (or **`docker compose`**) command to run containers and related operations according to the configurations in a YAML file

YAML (Yet Another Markup Language)

- Often used to describe software configurations in a human-readable way
- Use `[...]` for lists/arrays and `{...}` for name-value maps/dictionaries.
- Can be considered as a superset of JSON

Docker compose → used to manage many Docker files at a time

Docker Compose Procedure

- Define a Dockerfile for each image to be built
- Define needed services in a YAML configuration file (docker-compose.yml or compose.yaml)
- Run docker compose up

The docker compose command can specify which YAML configuration file to use by using -f option; e.g., docker compose –f any_yaml_file_name up

Inter-Process Communication (IPC)

Communication between Microservices

Actual communication happens between the executed software processes.

Enterprise applications often need to use each other's functions or services, and exchange data in order to realise business processes, between and within enterprises

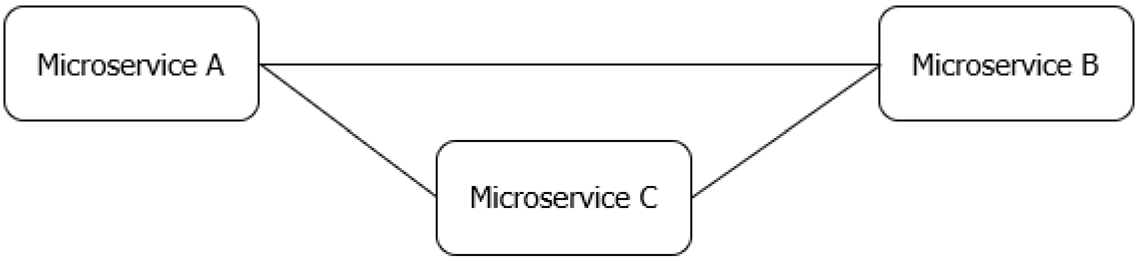
Categorisation Criteria for Communication Patterns

1) **How many receivers** does the sender expect **for the same data sent**?
    a) One receiver: one-to-one
        i) A sender can simulate one-to-many by repeating one-to-one many times (but it is still one-to-one)
    b) Many receivers: one-to-many
        i) All receivers: one-to-all
        ii) Selected receivers: one-to-selected (can be zero, one, multiple, or all) → e.g. email broadcast
2) Does the sender expect a **reply** for the data sent?
    a) No: fire-and-forget (fire & forget, one-way) → e.g. radio broadcast
    b) Yes: request-reply (request-response) → e.g. phone call, email
3) Do the sender and the receiver need to be **online at the same time** to realise the communication?
    a) Yes: synchronous (synchronized, sync.) → e.g. phone call, place order for a book, radio broadcast
    b) No: asynchronous (asynchronized, async.) → usually has some intermediary e.g. for email, the post office/email server is the intermediary, Grab App

Communication Technologies (aka Protocols)

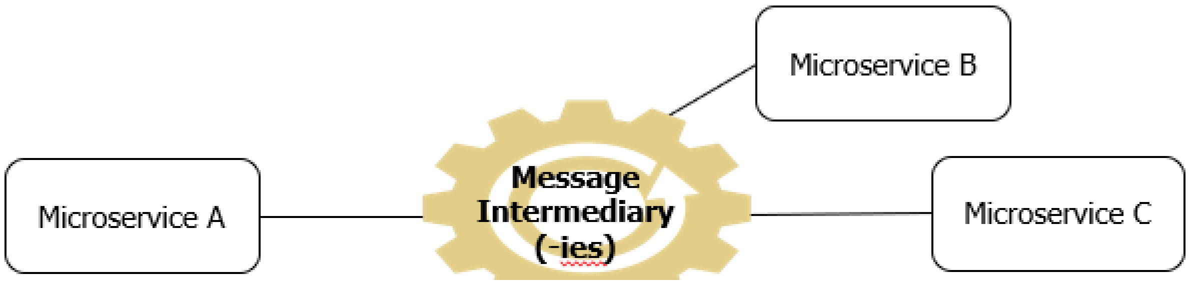
Used to implement various communication patterns

- **Invocation-based**: data goes **point-to-point**
  - Usually synchronous. E.g., function (method) call; **HTTP**; gRPC
  - Senders and receivers need to be online at the same time, so that they are more "tightly coupled"
  - In case of receiver failure or offline, no intermediary to store data and sender cannot send data
  - Sender needs to know all the receivers → More suitable for orchestration
  - Usually sender determines which receiver to receive data

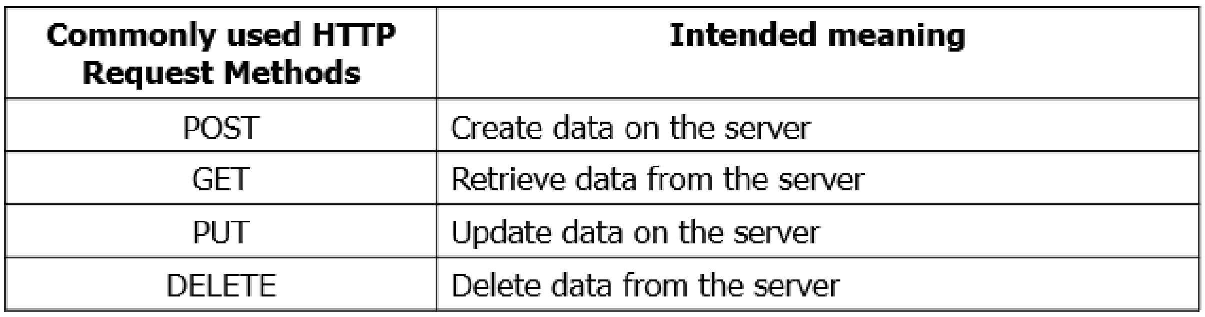


Phone call: a sender connects to a receiver by dialling a phone number; communication fails if the receiver doesn't answer
Web browsing (via HTTP): a web browser takes us to an URL; communication fails (e.g., 404) if the website doesn't respond properly.

- **Message-based** (a.k.a., **messaging**): data goes through a **message broker** or intermediary(-ies)
  - Usually asynchronous.
  - E.g., **AMQP**, JMS, IPFS, …
  - The intermediary can help to facilitate one-to-many communication
  - Much easier to do one-to-many than invocation-based
  - More loosely coupled between sender and receiver



HTTP Protocol



Commonly used HTTP Request Methods	Intended meaning
POST	Create data on the server
GET	Retrieve data from the server
PUT	Update data on the server
DELETE	Delete data from the server

We'll learn using python and its built-in "requests" module to send HTTP requests and receive responses.

- We can choose the method to be used by requests.request(…).
- E.g. requests.request(‘POST’, url, json = {})

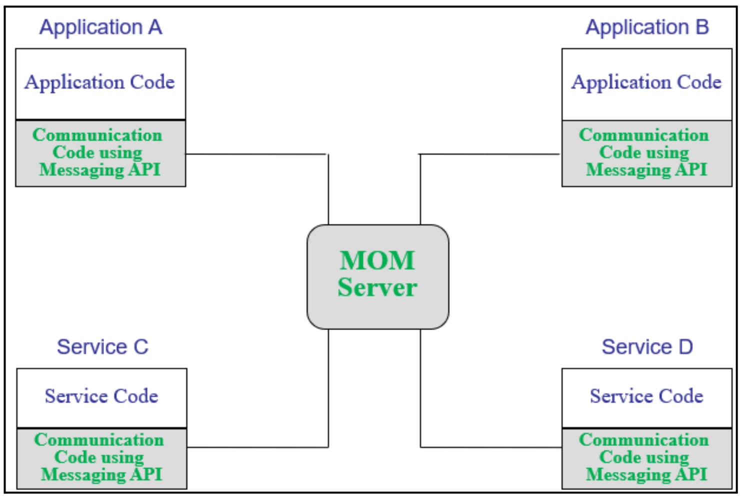
If a process or a user **needs an immediate response**, then the communication with all the relevant underlying microservices should use **invocation-based**, **synchronous** communication. → to make it straightforward in receiving a response and avoid the perception of no reaction or slow performance

If a process or a user **does not need an immediate response** (i.e., the process may be long-running or the user may interact at later time), then the communication with all the relevant underlying microservices should use **message-based**, **asynchronous** communication.

Communication Tech - Messaging

Message Oriented Middleware (MOM)

MOM is a software which sits between two or more applications or (micro)services and allow them to exchange data in the form of **messages**

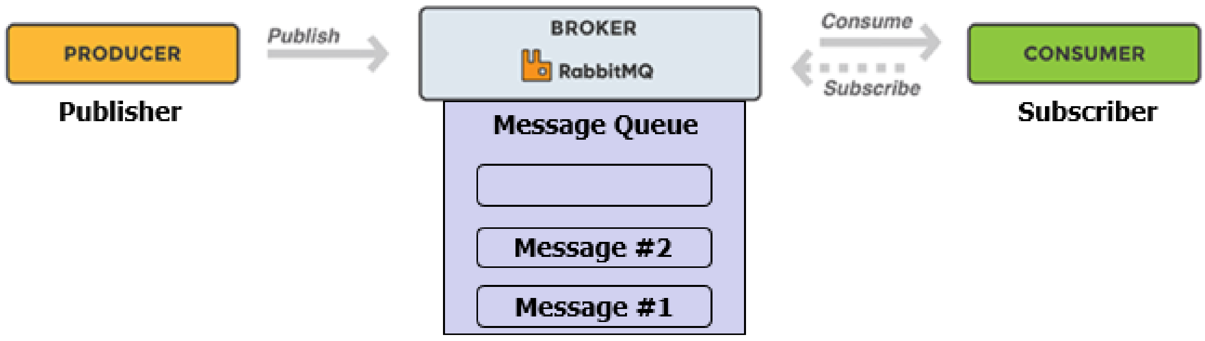

Application A
Application Code
Communication Code using Messaging API
Application B
Application Code
Communication Code using Messaging API
MOM Server
Service C
Service Code
Communication Code using Messaging API
Service D
Service Code
Communication Code using Messaging API

Advanced Message Queuing Protocol

- An **open standard protocol** for MOM
  - Define a message format that allows standard and extensible representations of various types of data
  - (In theory) Provide **interoperability** among different AMQP-compliant implementation software that can be used on either the client side or the server side
- Supports various communication patterns
  - One-to-one, one-to-many, fire-forget, request-reply, etc.
- RabbitMQ is an implementation of AMQP
  - pika is a python module for clients in python to interact with AMQP / RabbitMQ servers.

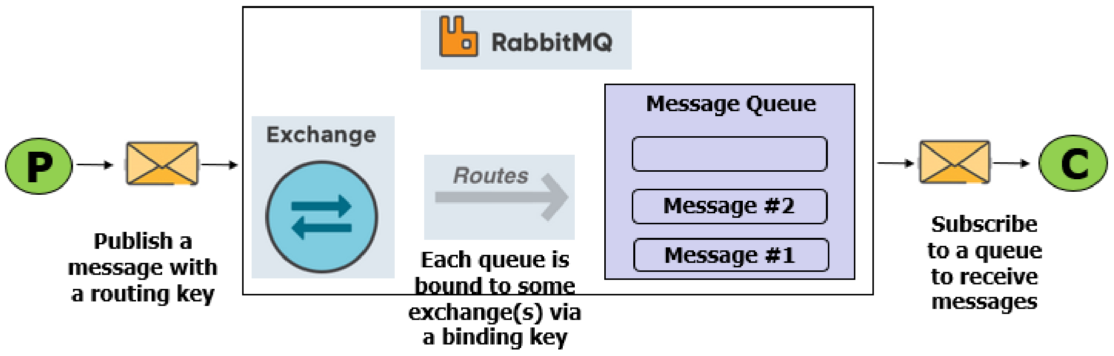
RabbitMQ

- **RabbitMQ** is a message-queueing software;
- It supports **AMQP** (and other protocols)
- It can be called a **message broker** or **queue manager**



Messaging Process

1. The **broker** accepts and forwards messages
2. A **producer** (publisher) sends messages to the broker
    a. A message can be anything e.g.
        i. Order message- {"order id":14, cust id:"191"}
        ii. Order event- "New order 14 created"
3. A **consumer** (subscriber) receives messages from the broker
4. The broker can keep a message in a **queue** until a consumer takes the message off the queue
    a. Many queues can be created
    b. Many messages can be sent to one queue; the same message can be sent to multiple queues
    c. A message is routed to a queue or queues via an exchange in the broker and key pattern matching


P
Publish a message with a routing key
RabbitMQ
Exchange
Routes
Each queue is bound to some exchange(s) via a binding key
Message Queue
Message #2
Message #1
Subscribe to a queue to receive messages
C

Types of Exchanges

- **Direct:** A direct exchange delivers a message to a queue whose **binding key** to the exchange **matches exactly** the **routing key** of the message sent to the exchange
- **Topic:** A topic exchange does a **wildcard match** between the routing key of a message and the binding keys of the queues bound to the exchange
    - I.e., a message is sent to a queue if the queue is bound to the exchange and the routing key of the message matches the wildcard pattern of the binding key of the queue
- **Fanout:** routes a message to **all** of the queues that are bound to it

A queue in RabbitMQ has to be bound to an exchange in order to receive a message

The sender/publisher does not need to subscribe to a queue in order to send a message.

- However, the consumer/receiver has to subscribe to a queue in order to receive a message.

A message producer (sender) in RabbitMQ can be a service consumer; a message consumer (receiver) in RabbitMQ can be a service provider

To determine whether routing key and binding key match:

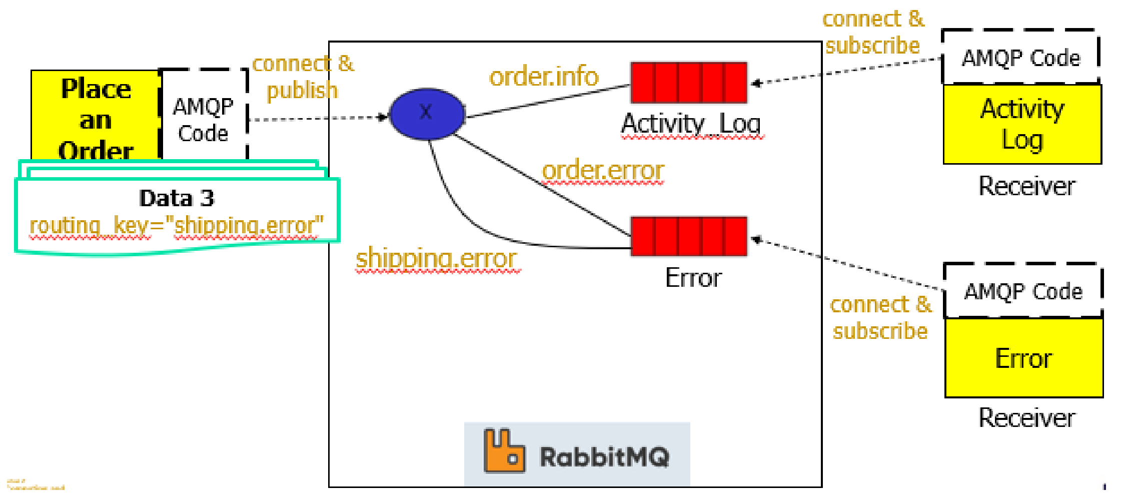
A key is a sequence of words separated by ".“

* matches any one word; # matches zero, one, or many words;

Topic exchanges with proper binding keys can be used to implement the same functionality of direct / fanout exchanges too.

Implementations: Bookstore Scenario

Implementation: AMQP with *Direct* Exchange



The **sender** doesn't need to know any receiving queue's name, as long as it knows the right routing key to use (in this case, "shipping.error")

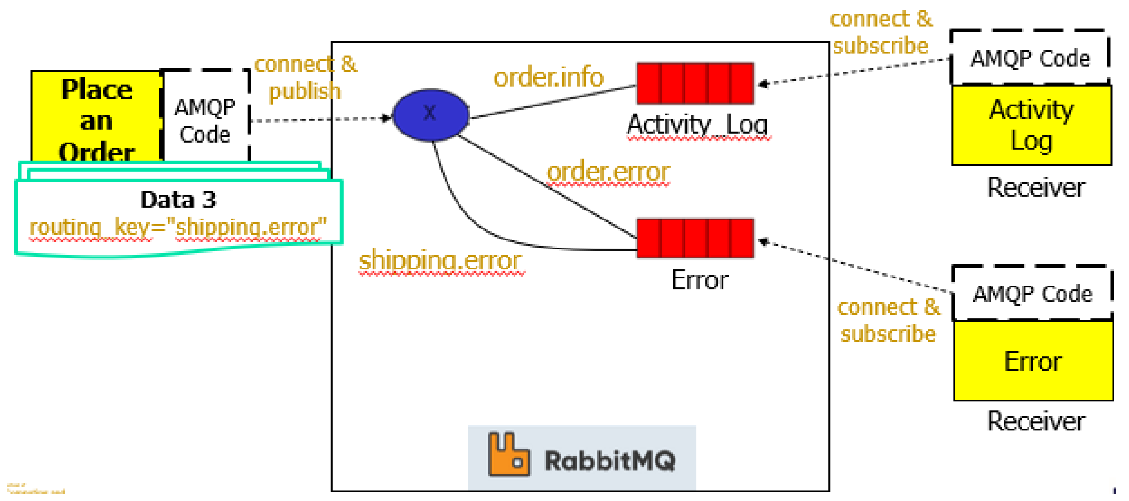
A key is a sequence of words separated by ".", e.g., order.info, order.error

The **exchange** in the **broker** routes the message to the **queue**(s) with the matching **binding key**. ("shipping.error")

A **receiver** consumes the **message**(s) routed to the **queue**(s) that it is consuming.

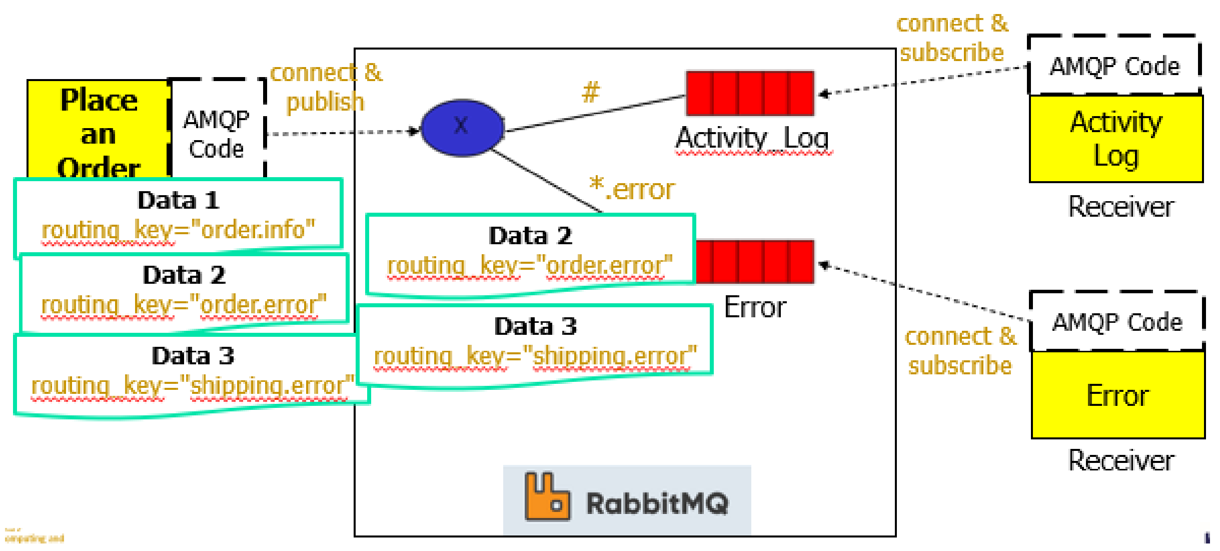
- In this example, only error microservice receives the message, activity log microservice does not receive

Implementation: AMQP with *Fanout* Exchange



- The queue(s) are bound to a fanout exchange with no binding key;
- Every queue bound to the exchange gets every message sent to the exchange;
- In this case, both activity log microservice and error microservice receive the message

Implementation: AMQP with *Topic* Exchange



The **wildcards** are in the binding key.

* matches any *one* word in the keys;

- *.error matches order.error, shipping.error, but not other.kinds.of.error;

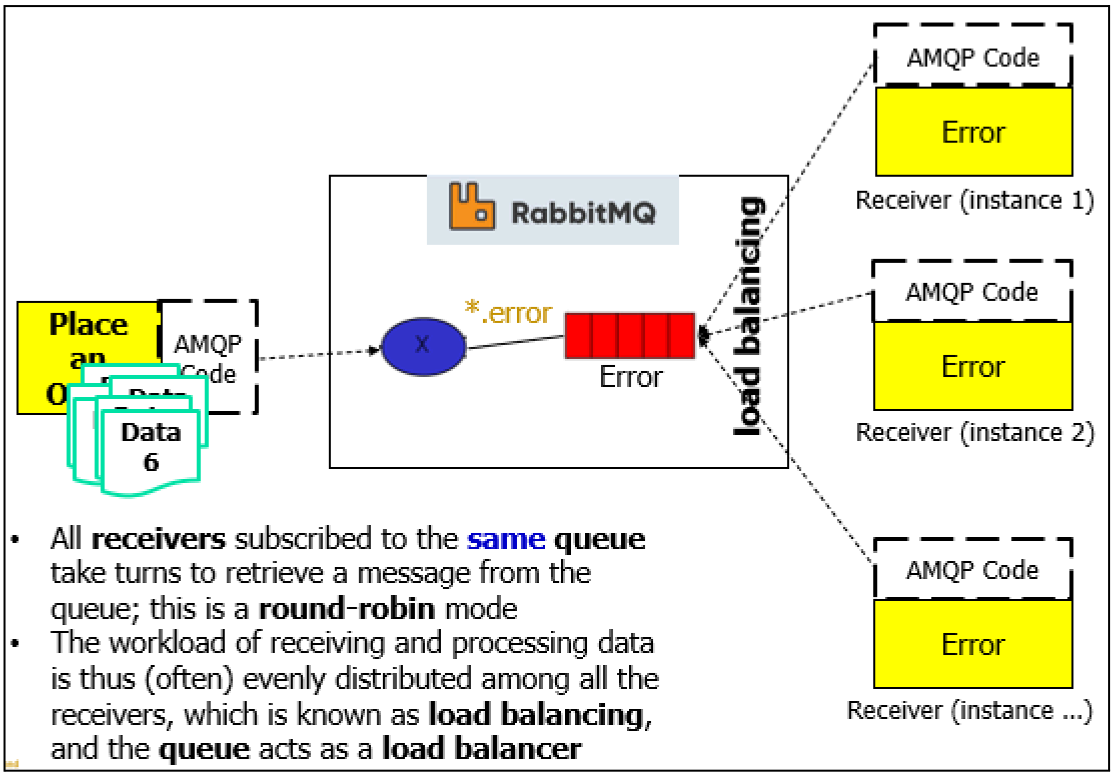
# matches zero, one, or many words.

- #.info matches info, order.info, more.order.info, or more.and.more.info;
- # matches any key.

In the example above:

- order.info gets routed to activity log
- order.error gets routed to both error (correct) and activity log (# binding key = anything)
- shipping.error gets routed to both error (correct) and activity log (# binding key = anything)

Implementation: Load Balancing



- All **receivers** subscribed to the **same queue** take turns to retrieve a message from the queue; this is a **round-robin** mode
- The workload of receiving and processing data is thus (often) evenly distributed among all the receivers, which is known as **load balancing**, and the **queue** acts as a **load balancer**

- All receivers subscribe to the same queue and take turns to retrieve a message from the queue
    - Sometimes the queue may use a random message distribution strategy, instead of a strict round-robin mode. Then, each message may go to a random receiver.
- The workload of receiving and processing data is thus (often) evenly distributed among all the receivers, which is known as **load balancing**, and the **queue** acts as a **load balancer**