Typical Microservices Architecture

Concept

A **microservices architecture** breaks a large monolithic application into smaller, independent services that communicate via APIs.
Each service handles a **specific business capability** (e.g., authentication, payments, catalog, orders).

Core Components

Component	Description
API Gateway	Acts as the single entry point to all backend microservices. Handles routing, authentication, rate limiting, caching, and protocol translation.
Service Registry / Discovery	Keeps track of active service instances and enables dynamic service discovery. Tools: Consul, Eureka, etcd.
DNS	Maps service names to IPs for internal and external access. Often integrated with service discovery for microservices.
Load Balancer	Distributes incoming traffic across instances of a service to ensure high availability and scalability. Can be hardware-based, software-based, or cloud-managed (e.g., Nginx, AWS ELB).
Database per Service	Each microservice owns its database to maintain loose coupling. Can be SQL (PostgreSQL, MySQL) or NoSQL (MongoDB, DynamoDB, Cassandra) depending on requirements.
SQL Databases	Structured data storage, supports transactions and complex queries. Best for relational data and consistency needs.
NoSQL Databases	Flexible schema, horizontal scalability, high throughput. Suitable for unstructured data, caching, and fast lookups.
Synchronous Communication	Request-response communication between services, typically over HTTP/REST or gRPC. Easy to implement but can increase latency.
Asynchronous Communication	Event-driven communication via message brokers like Kafka, RabbitMQ, or AWS SQS. Decouples services and improves scalability and resilience.
	Handles async communication, message queues, pub/sub, and event

Message Broker	streaming. Examples: Kafka, RabbitMQ, AWS SQS.
Swagger / OpenAPI	API documentation and contract generation. Helps teams understand endpoints, request/response models, and test APIs easily.
Event Sync / Event Bus	Services publish events to an event bus (Kafka, NATS). Other services subscribe to events for reactive workflows. Supports eventual consistency.
Externalized Config & Logs	Centralized configuration (e.g., Spring Cloud Config, Consul) and externalized logs (ELK stack, CloudWatch) for easier management and troubleshooting.
Monitoring / Tracing	Observability tools to track service health, performance, and errors. Examples: Prometheus, Grafana, Jaeger, Zipkin.
Reporting / Analytics	Aggregates data from multiple services for business intelligence and reporting. Often built on ELK, Redshift, Snowflake, or custom analytics pipelines.

2. DNS (Domain Name System)

Purpose

DNS translates **human-readable domain names** (like `myapp.com`) into **IP addresses** (like `192.168.1.2`).

How It Works

1. **Client requests** `myapp.com`
2. DNS resolver queries:
    - **Root Server → TLD Server (.com) → Authoritative Server**
3. Returns IP address (e.g., `54.12.32.18`)
4. Browser connects to that IP.

In Cloud Context

Services like **AWS Route 53**, **Azure DNS**, or **Cloudflare** manage DNS zones with load

balancing, failover, and latency-based routing.

---

3. Load Balancers

Purpose

A **load balancer (LB)** evenly distributes incoming requests among multiple backend servers to ensure:

- High availability
- Fault tolerance
- Scalability

Types of Load Balancers

Type	Operates On	Example	Use Case
L4 (Transport)	TCP/UDP	AWS NLB, HAProxy	Fast, for network-level routing
L7 (Application)	HTTP/HTTPS	AWS ALB, Nginx, Traefik	Smarter, inspects URLs, headers, cookies

Features

- Health checks
- Sticky sessions
- SSL termination
- Auto-scaling integration

---

4. Synchronous vs Asynchronous Communication

Aspect	Synchronous	Asynchronous
Definition	Caller waits for response	Caller sends message and continues
Example	REST API call	Message queue (Kafka, RabbitMQ)
Protocol	HTTP/HTTPS	AMQP, MQTT, Kafka
Use Case	Real-time responses (e.g., login)	Decoupled processing (e.g., notifications, billing)
Drawback	Tight coupling, latency-sensitive	Complex to debug, eventual consistency

Visual:

```
SYNC:  Client → Service A → waits → Response
ASYNC: Client → Service A → sends to Queue → Service B processes later
```

---

5. SQL vs NoSQL Databases

Feature	SQL (Relational)	NoSQL (Non-relational)
Structure	Tables, rows, columns	Key-value, Document, Graph, Wide-column
Schema	Fixed	Dynamic
Scaling	Vertical (add more CPU/RAM)	Horizontal (add more nodes)
Examples	MySQL, PostgreSQL	MongoDB, DynamoDB, Cassandra
Use Cases	Transactions, structured data	High scalability, flexible schema
Query Language	SQL	Custom (JSON-like)

Microservices often use both, depending on the service's need (polyglot persistence).

---

6. API Gateway

Role

An **API Gateway** sits between the client and backend services.

Responsibilities

Feature	Description
Routing	Direct requests to proper microservice
Authentication / Authorization	Validate tokens (OAuth2, JWT)
Rate Limiting & Throttling	Protect from overload
Request Transformation	Modify headers, URLs, or payloads
Caching	Improve response speed
Monitoring / Logging	Centralized request tracing

Popular Gateways

- AWS API Gateway
- Kong
- Nginx
- Istio (in service mesh)
- Apigee (Google Cloud)

---

7. Externalizing Logs

Why Externalize?

Each microservice runs in containers/pods — their local logs vanish when the container restarts.
So logs are **centralized** in an external system.

Common Setup

Tool	Role
Fluentd / Fluent Bit / Logstash	Collect logs
Elasticsearch / Loki	Store & index logs
Kibana / Grafana	Visualize & search logs
Cloud-native Dynatrace	CloudWatch (AWS), Stackdriver (GCP)

Benefits

- Unified view of system behavior
- Easier debugging & auditing
- Log-based alerting (via Prometheus + Alertmanager)

---

8. 12-Factor Apps

A **design philosophy** for building scalable, cloud-ready microservices.

Factor	Description
1. Codebase	One codebase tracked in version control, many deploys
2. Dependencies	Explicitly declare via dependency manager (pip, npm)
3. Config	Store in environment variables, not code
4. Backing Services	Treat DBs, queues, caches as attached resources
5. Build, Release, Run	Strictly separate stages
6. Processes	Execute app as stateless processes
7. Port Binding	Expose services via port binding
8. Concurrency	Scale out via process model
9. Disposability	Fast startup/shutdown
10. Dev/Prod Parity	Keep environments similar
11. Logs	Treat logs as event streams
12. Admin Processes	Run admin tasks as one-off processes

These principles ensure portability, scalability, and resilience — ideal for microservices, containers and Kubernetes.