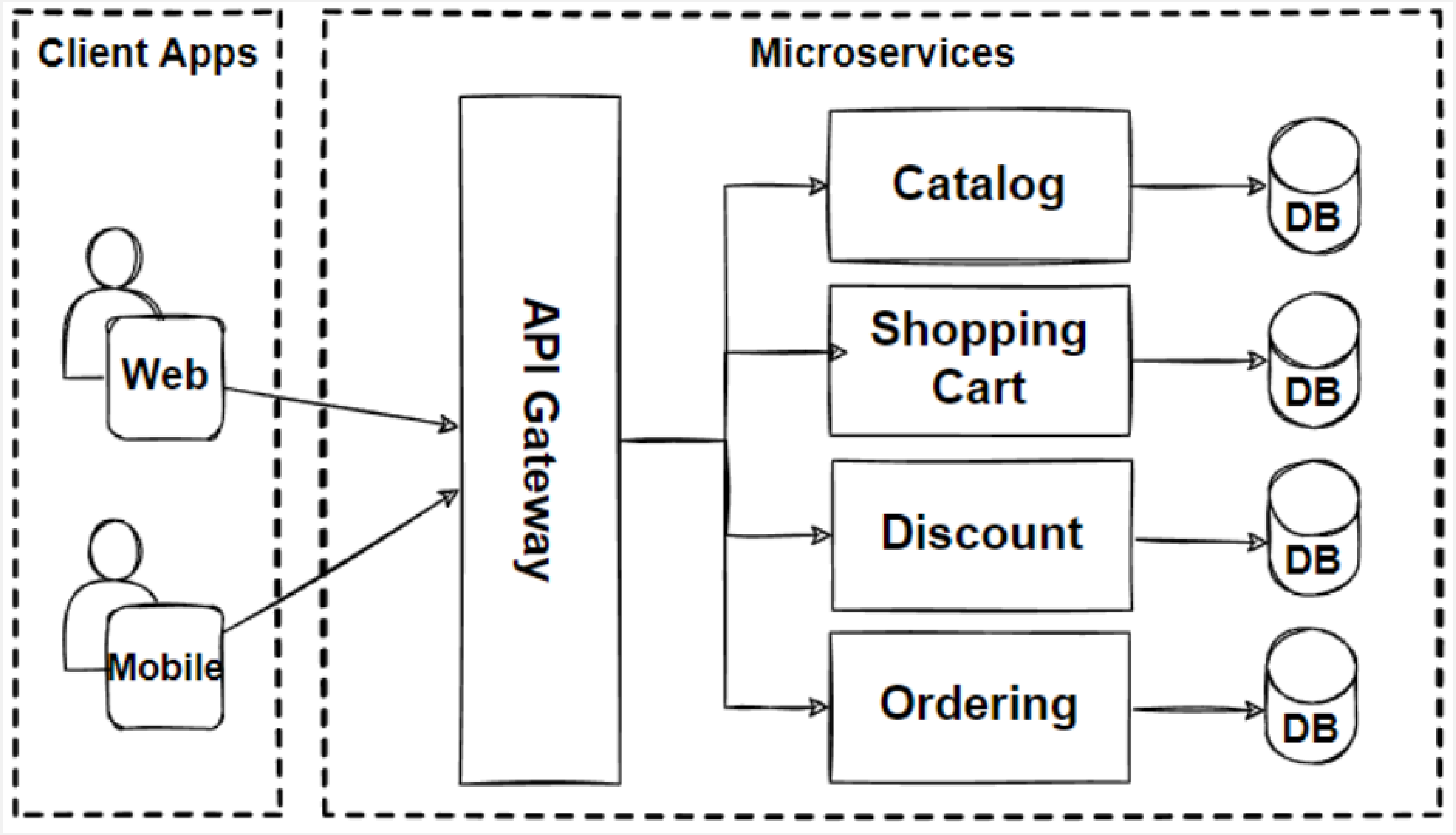
MICRO SERVICES

WHAT ARE MICROSERVICES

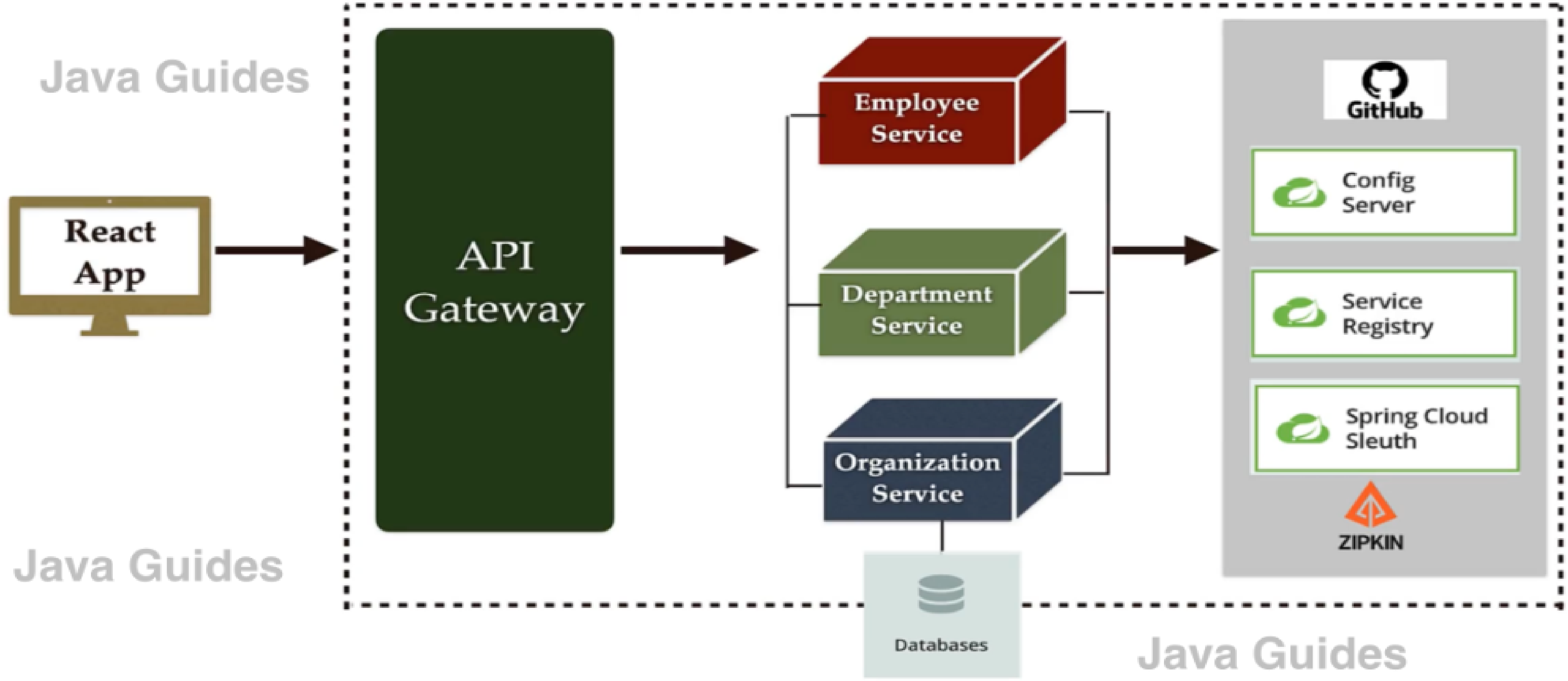
- Microservices are a software architectural style in which a large application is built as a collection of small, independent services that communicate with each other over a network.
- Each service is a self-contained unit of functionality that can be developed, tested, and deployed independently of the other services. This allows for more flexibility and scalability than a monolithic architecture, where all the functionality is contained in a single, large codebase.
- Microservices can be written in different programming languages and use different technologies, as long as they can communicate with each other through a common API.
- They are designed to be loosely coupled, meaning that changes to one service should not affect the other services. This makes it easier to update, maintain, and scale the application. Microservices architecture is best suited for large and complex applications that need to handle a high volume of traffic and need to be scaled horizontally.


Client Apps
Microservices
Web
Mobile
API Gateway
Catalog
Shopping Cart
Discount
Ordering
DB
DB
DB
DB

- Key components of a microservices architecture include:

1. **Core Services**: Each service is a self-contained unit of functionality that can be developed, tested, and deployed independently of the other services.
2. **Service registry**: A service registry is a database of all the services in the system, along with their locations and capabilities. It allows services to discover and communicate with each other.
3. **API Gateway:** An API gateway is a single entry point for all incoming requests to the microservices. It acts as a reverse proxy, routing requests to the appropriate service and handling tasks such as authentication and rate limiting.
4. **Message bus:** A message bus is a messaging system that allows services to communicate asynchronously with each other. This can be done through protocols like HTTP, RabbitMQ, or Kafka.
5. **Monitoring and logging:** Monitoring and logging are necessary to track the health of the services and troubleshoot problems.
6. **Service discovery and load balancing:** This component is responsible for discovering service instances and directing traffic to the appropriate service instances based on load and availability.
7. **Continuous integration and continuous deployment (CI/CD):** To make the development and deployment process of microservices as smooth as possible, it is recommended to use a tool such as Jenkins, TravisCI, or CircleCI to automate the process of building, testing, and deploying microservices.

Microservices Architecture using Spring boot and Spring Cloud



SPRING CLOUD

- Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer’s own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.

FEATURES

- Spring Cloud focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others.
- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Global locks
- Leadership election and cluster state
- Distributed messaging

CONFIG-SERVER TO EXTERNALIZE THE CONFIGURATIONS

- Next, we will implement a config server to externalize the configurations of all these three microservices into a central place which is the git repository.
- Spring cloud provides a Spring cloud config module that we can use to implement a config server to externalize the configuration files of all these three microservices into a central place. We are going to use the git repository as storage for the config server.

NETFLIX OSS

- Netflix OSS is a set of frameworks and libraries that Netflix wrote to solve some interesting distributed-systems problems at scale. Today, for Java developers, it's pretty synonymous with developing microservices in a cloud environment. Patterns for service discovery, load balancing, fault-tolerance, etc are incredibly important concepts for scalable distributed systems and Netflix brings nice solutions for these

CORE MICROSERVICES

- Consider we have developed three core backend Spring boot microservices such as **employee service**, **department service,** and **organization service,** and all these three microservices have their own databases. You can use a relational database or NoSQL database as a database for these microservices. So whenever you create a microservice in your project, make sure that each microservice should have its own database. All right.

MICROSERVICES COMMUNICATION

- Once we build these 3 microservices. Next, we'll see how these microservices communicate with each other. Well, there are different ways to make a REST API call from one microservice to another Microservice. For example, we can use a RestTemplate or WebClient or Spring cloud-provided open feign library
- here are two types of communication styles. One is synchronous and another is asynchronous.

  In the case of synchronous, we can use the HTTP protocol to make an HTTP request from one microservice to the microservice.

  And in the case of asynchronous communication, we have to use a message broker for asynchronous communication between multiple microservices. For example, we can use RabbitMQ or Apache Kafka as a message broker in order to make an asynchronous communication between multiple microservices and each microservice in a microservices project can expose REST APIs.

REGISTRY AND DISCOVERY PATTERN

- once we know how microservices communicate with each other, next you need to know how to implement a service Registry and discovery pattern in our microservices project.
- Spring Cloud provides a Spring Cloud Netflix Eureka Based Service Registry module that we can use to implement service registry and discovery patterns in our microservices project. Well, service Registry and discovery is a really essential pattern that we can use to avoid hard coding hostnames and ports.

API GATEWAY PATTERN

- once we know how to use the config server to externalize the configuration files. Next, we have to implement an API gateway.
- API Gateway plays a very important role in our microservices architecture. So whenever a client wants to make a call to different microservices, the client has to remember the host names and ports of all these microservices. So there should be a solution where a client can send a request to the central component so that is where the API gateway comes into the picture. So whenever a client sends a request to the backend microservices, then the client has to send a request to the API gateway first, and then the API gateway based on the routing rules will route that request to the appropriate microservice. So this is how the API gateway plays an important role in a microservices architecture.
- Spring Cloud provides **Spring Cloud Gateway module** to implement API gateway patterns in a microservices architecture.

DISTRIBUTED TRACING

- Next, once you know how to implement an API gateway in a microservices project, next you can implement distributed tracing in a microservices architecture. Well, Spring Cloud provides a Spring Cloud sleuth module, which we can use to implement distributed tracing in our microservices project.
- along with Spring Cloud Sleuth, we'll also use Zipkin to visualize the tracing log information in a user interface. Well, Zipkin provides a user interface to track and trace information through web applications.
- Next, you can use React/Angular to create a client-side service that will make a call to backend microservices.

CIRCUIT BREAKER PATTERN

- you can implement a circuit breaker pattern in an employee service because the employee service is internally calling department service, and let's say due to some reason, department service is down then employee service won't get a response from the department server, isn't it? And then again, employee service will send an internal server error to the API gateway and then API Gateway will send that response back to the client. All right. So in order to avoid this kind of issue, we can use a circuit breaker pattern.

  So this circuit breaker pattern helps the employee service to avoid continuous calls to the department service Whenever department service is done and this circuit breaker pattern will help employee service to return some default response back to the API Gateway and the API Gateway will send that default response to the client.