**Synchronous Communication:**

In microservices, it refers to a communication pattern where a service sends a request to another service and waits for a response before continuing its execution.

**Asynchronous Communication:**

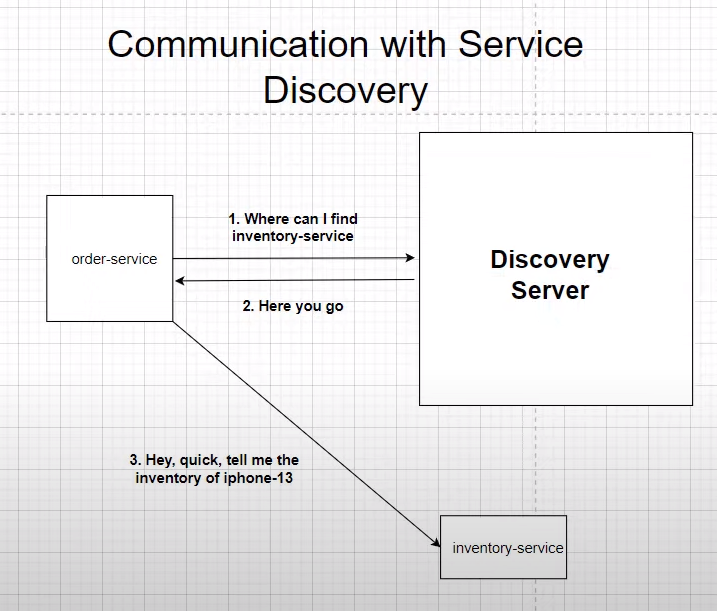
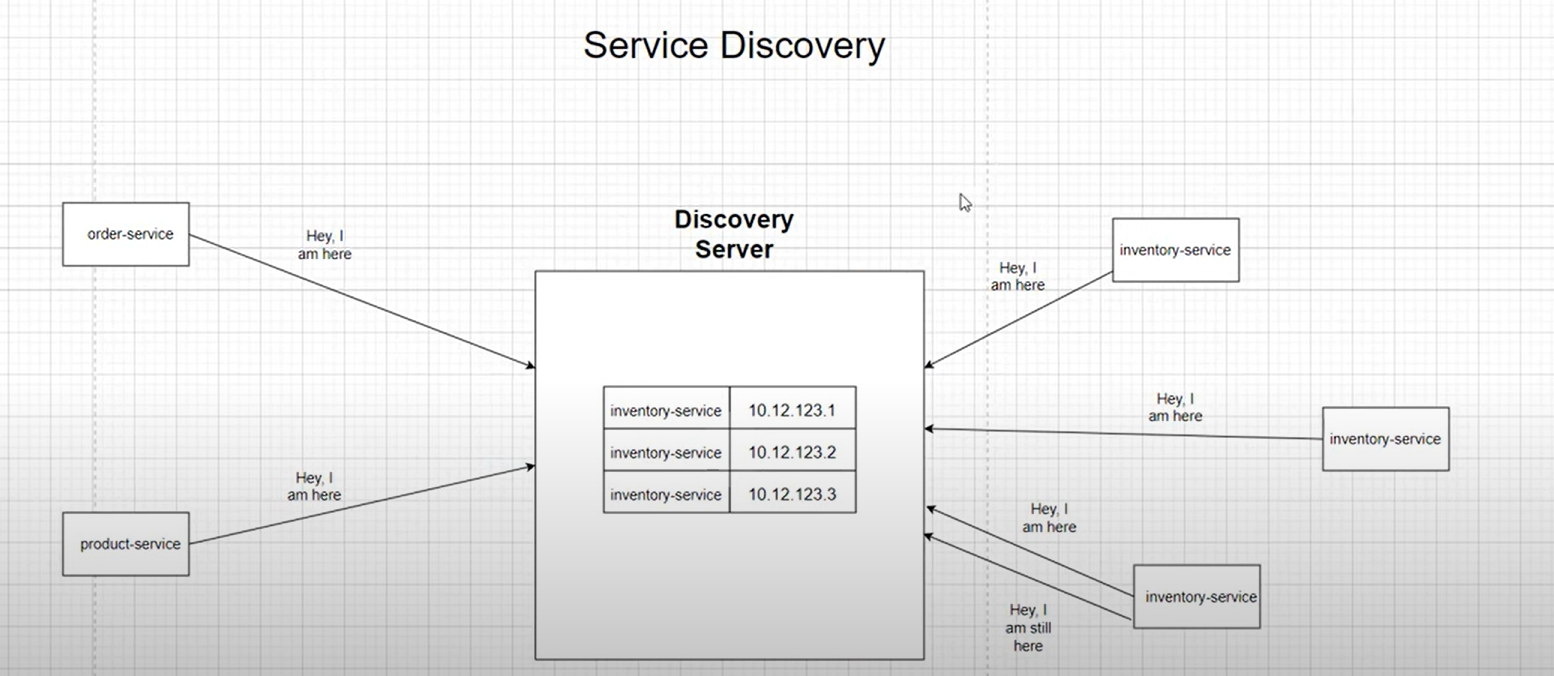
It allows microservices to communicate without waiting for an immediate response, enabling more decoupled and resilient systems.

**Webclient:**

* Alternate to RestTemplate but advanced.
* By default, it makes asynchronous communication.

**Service Discovery:**

* Stores info about services (service name & ip address)
* Services register with the discovery server by making a request.
* Discovery server adds the entries of the services in its local copy called **Register.**
* Then service make a call to discovery server for dependency service, then discovery server provides necessary info about target service.

****

* When making a initial call to discovery server, the discovery server sends registry as the response to the client, the client stores the copy in a separate location.
* If discovery server not available it first checks the local copy and get info about service.

A diagram of a service

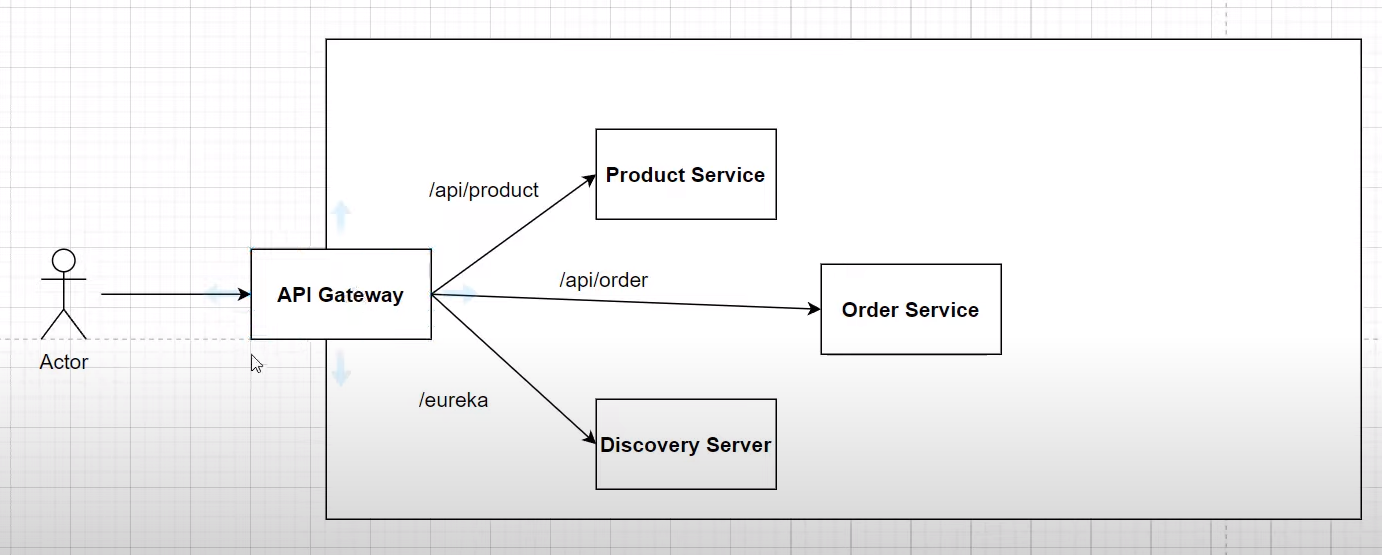
Description automatically generated

* **Dependency needed:**
  + Eureka server for Discovery server.
  + Spring cloud depedency management dependency
  + **Dependency management should be in root pom.**
  + **@EnableEurekaServer should be added in main class of the discovery service**
  + **Eureka server default port = 8761**
  + **Properties**
    - **eureka.instance.hostname=localhost  
      eureka.client.register-with-eureka=false  
      eureka.client.fetch-registry=false  
        
      server.port=8761**
  + after adding properties, now add eureka client dependency to service.
  + **Then add @EnableEurekaClient in main of every service. (**this annotation in not mandatory & is under deprecation, which means client dependency is enough**)**
    - eureka.client.service-url.defaultZone=http://localhost:8761/eureka
    - **for latest version no need to add above property in each service (eureka server automaticcaly detected by services)**
  + **Then in each service add properties of eureka client.**

**Feign Client:**

* it helps to make synchronous communication among services.
* Add openfeign dependency, enable @EnableFeignClients in main class.
* Now create a feign client (interface) using @FeignClient and add the necessary dependent endpoints in it.
* It is only used to make synchronous requests, to make asynchronous requests use webflux

**API Gateway:**

****

* We can configure **API Gateway** based on,
  + Routing based on Request Headers
  + Authentication
  + Security
  + Load Balancing
  + SSL Termination
* This is the place where all requests pass through (entry point for all api requests)
* Below properties means, if the client sends the requests to **/api/product** then route it to **product-service.**
  + **There are different kinds of Routes to configure, following is one of the type**

#Product-Service-Routes  
spring.cloud.gateway.routes[0].id=product-service  
spring.cloud.gateway.routes[0].uri=lb://product-service  
spring.cloud.gateway.routes[0].predicates[0]=Path=/api/product

* **Access Eureka Server thrugh gateway with custom endpoint**

#Discovery-server-Routes  
spring.cloud.gateway.routes[1].id=discovery-server  
spring.cloud.gateway.routes[1].uri= http://localhost:8761  
spring.cloud.gateway.routes[1].predicates[0]=Path=/eureka/web  
spring.cloud.gateway.routes[1].filters[0]=SetPath=/  
  
#Discovery-server-static resources route  
spring.cloud.gateway.routes[2].id=discovery-server-static   
spring.cloud.gateway.routes[2].uri= http://localhost:8761  
spring.cloud.gateway.routes[2].predicates[0]=Path=/eureka/\*\*

**Notes:**

* **Maven Management:**
  + Parent of root pom should be springboot.
  + All common dependencies, builds, dependency management can be added in root pom.
  + To verify everthing is correctly added run maven goal **mvn clean verify.**
* For running multiple instances of a service, don’t hardcode port number In properties file.
  + put **server.port =0** **,** so springboot can run the app in a random port
  + always run all services on a random port.
* **spring.application.name = product-service**
* **It takes time to register services, so after starting service wait for 30sec.**
* **@LoadBalanced for client side loadbalancing**
* **Even if discovery server is stopped, the client will store the local copy of registry. So, there won’t be an issue.**
* **Transport Exception -> if service not able to detect discovery server then it throws this exception**

**Circuit Breaker:**

* Is mainly used to have resilent communication between services.
* It is a mechanism to detect & handle failures in a way that prevents them from cascading throughout the system.
  + Parent of root pom should be springboot.
* It is used in both synchronous & asynchronous communication.
* Netflix Hysterix, Resilience4j are some libraries to implement circuit breaker.
* Resilience4j is inspired from hysterix.
* States of circuit breaker,
  + Closed - if everything is working fine.
  + Open - if something gone wrong with services (can execute some fallback logic)
  + Half open - once the issue is fixed & circuit breaker is put into half open to check whether everything is working fine or not. If it is then state changed to **closed.**
* Communication among services,
  + **Synchronous Communication:**
    - Here, a client sends request to a service & waits for a response.
    - If the service being called experiences issues (slow response time, errors), the circuit breaker can be triggered to stop sending additional requests to that service for a period.
    - During this **open** state, the circuit breaker can provide a fallback response to the client instead of waiting for the problematic service to respond.
  + **Asynchronous Communication:**
    - Here, a client sends request to a service, but it doesn’t wait for a immediate response.
    - If the service has a circuit breaker & experiences issues, it can take similar actions by stopping processing of requests for a period.
    - During this time, the service might handle the situation by storing the messages in a queue/applying a fallback mechanism to process them differently.
* Dependency,

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>  
</dependency>

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

* Resilience4j properties

resilience4j.circuitbreaker.instances.inventory.registerHealthIndicator=true  
resilience4j.circuitbreaker.instances.inventory.event-consumer-buffer-size=10  
resilience4j.circuitbreaker.instances.inventory.slidingWindowType=*COUNT\_BASED*resilience4j.circuitbreaker.instances.inventory.slidingWindowSize=5  
resilience4j.circuitbreaker.instances.inventory.failureRateThreshold=50  
resilience4j.circuitbreaker.instances.inventory.waitDurationInOpenState=5s  
resilience4j.circuitbreaker.instances.inventory.permittedNumberOfCallsInHalfOpenState=3  
resilience4j.circuitbreaker.instances.inventory.automaticTransitionFromOpenToHalfOpenEnabled=true

#Resilience4J Timeout Properties  
resilience4j.timelimiter.instances.inventory.timeout-duration=3s

* /actuator/health - actuator endpoint for health
* /actuator - lists all endpoints
* Implementation,

@PostMapping  
@ResponseStatus(HttpStatus.*CREATED*)  
@CircuitBreaker(name = "inventory", fallbackMethod = "fallbackMethod")  
@TimeLimiter(name = "inventory")  
@Retry(name = "inventory")  
public CompletableFuture<String> placeOrder(@RequestBody OrderRequest orderRequest) {  
 return CompletableFuture.*supplyAsync*(() -> orderService.placeOrder(orderRequest));  
}  
  
public CompletableFuture<String> fallbackMethod(OrderRequest orderRequest, RuntimeException runtimeException) {  
 return CompletableFuture.*supplyAsync*(() -> "oops something went wrong, please order after some time");  
}

**Distributed Tracing:**

* It is a techinique used in microservices architectures to monitor & troubleshoot the flow of requests as they traverse through various services.
* It provides a way to trace the path of a request as it moves through different components of a distributed system.
* This is particularly important in microservices environments, where apps are composed of multiple loosely coupled services.
* Sleuth is a distributed tracing framework provided by spring.
  + But this is moved to Micrometer Tracing & is the latest.
* Zipkin is UI tool used to visualize the request information.
* Flow of request,
  + TraceId is unique id for whole request.
  + SpanId is the no. of trips the request takes.
    - Has unique id for each trip.
    - In the middle of a request, if another thread is involved then it can be counted as a new request, so It won’t be reflected in zipkin UI. To sort out this issue, we can **create custom spans**, where we can track the request from start to finish, even it works with multiple threads.
    - If we are making a call, I a single thread then we can trace it from start to end.

A diagram of a distributed tracing

Description automatically generated

* Dependencies needed,

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>  
<dependency>  
 <groupId>io.micrometer</groupId>  
 <artifactId>micrometer-tracing-bridge-brave</artifactId>  
</dependency>  
<dependency>  
 <groupId>io.zipkin.reporter2</groupId>  
 <artifactId>zipkin-reporter-brave</artifactId>  
</dependency>

* Properties to add,

# zipkin sampling properties  
management.tracing.sampling.probability=1.0

* Now, download zipkin server & run it using ***java -jar zipkin-server-2.12.9-exec.jar***
* Or we can run docker image of zipkin-server.
* Zipkin default port = 9411, we can access it at **http://localhost:9411/zipkin/**
* For sleuth, we may encounter compatible issues, so use following property to solve it temporarily. Don’t use it in production.

spring.cloud.compatibility-verifier.enabled=false

* After configuring everything send requests to server, so that the requests will be reflected in zipkin UI.

**Event Driven Architecture using Kafka:**

* EDA is a s/w design pattern in which the flow of the application is determined by events, such as user actions, sensor outputs, or messages from other programs/services.
* In EDA system, components/microservices communicate primarily through events, & they can be loosely coupled, allowing for greater flexibility & scalability.
* EDA basically converts a synchronous request to asynchronous request using events.
* EDA is most powerful in distributed systems, microservices & scenarios where asynchronous communication & reponsiveness to changins conditions are essential.
* EDA promotes flexibility, scalability & easier maintenance of complex systems by allowing components to communicate in a loosely coupled manner based on events.
* Key components & concepts of EDA,
  + Events
  + Event Producer
  + Event Consumer
  + Event Bus/Message Broker
  + Event Handler
  + Publish-Subscribe Model
  + Event-driven Microservices
  + CQRS (Command Query Responsibility Segregation)
  + Event Sourcing

**Telusko:**

**Blue-Green Deployment:**

* 2 instances are running but only one is live.
* Deploy the latest version in other instance, which is not live.
* Now, gradually move users from old version instance to latest version instance.
* After transferring all traffic now update the idle one to latest version.
* And we repeat the same process for each & every deployment.

**Cloud Ready:**

* Making non cloud app to support cloud services is called Cloud Ready application.

**Cloud Native:**

* Developing a project from scratch with the intention of leveraging cloud computing principles & services.

**12-Factor App:**

* **Codebase -** one code base tracked in version control, with multiple deployments.
* **Dependencies -** explicitly declare & isolate dependencies. Use a dependency management tool.
* **Config -** store configuration in the environment. Avoid storing it in the code.
* **Backing Services -** treat backing services (databases, message queues, etc.) as attached resources.The code should be able to switch b/w different services without changes.
* **Build, Release, Run -** strictly separate build & run stages. Keep builds & releases immutable.
* **Processes -** execute the app as one or more stateless processes. Avoid storing session state locally.
* **Port Binding -** export services via port binding. The app should be self-contained & able to run on any available port.
* **Concurrency -** scale out via the process model. Design the app to scale horizontally by adding more processes.
* **Disposability -** maximize robustness with fast startup & graceful shutdown. Aim for quick startup & graceful termination of processes.
* **Dev/Prod parity -** keep development, staging & prod environments as similar as possible. Use CI/CD practices.
* **Logs -** treat logs as event streams. Log messages should be treated as event streams, & logs should be accessible by developers.
* **Admin Processes -** run admin/management tasks one-off processes. Admin tasks should be run in the same environment as the application but as separate processes.

**Feign Client:**

* Used for service-service communication.
* Import **OpenFeign** dependency & annotate main class with **@FeignClient(“service\_name”)**
* Annotate main class with **@EnableFeignClients** annotation.
* Now, create a package ‘feign’ in service, where we are making requests.
* Now, create an interface, which acts as communication bridge between source & destination services.
* Now, add apis of destination service without body.
* Don’t forget to add whole endpoint in the above interface

**Service-Discovery:**

* Add eureka server to a new service.
* Configure properties.
* Add discovery client in client services.

**Load Balancer:**

* No need to add any dependency for load balancer.
* **FeignClient** automatically does load balancing, so no need for configuration.
* This is called **client-side load balancing.**

**API Gateway:**

* Add eureka discovery client, api gateway dependencies
* In properties file, add

spring.cloud.gateway.discovery.locator.enabled=true

**Notes:**

* To make 2 services independent, make sure one service don’t access the db of other service. (loose coupling)
  + i.e., the requested data should be sent as response to other service but don’t give access to the database.
* @ManyToMany is only for entities
  + **@ManyToMany**
  + **Private List<Course> courses;**
* But for integers, use
  + **@ElementCollection**
  + **Private List<Integer> courseIds;**

**After completing project do this:**

* change ports in configuration
* change routing from localhost to service