JAVA FULL STACK DEVELOPMENT PROGRAM

Session 27: Microservices and Spring Cloud

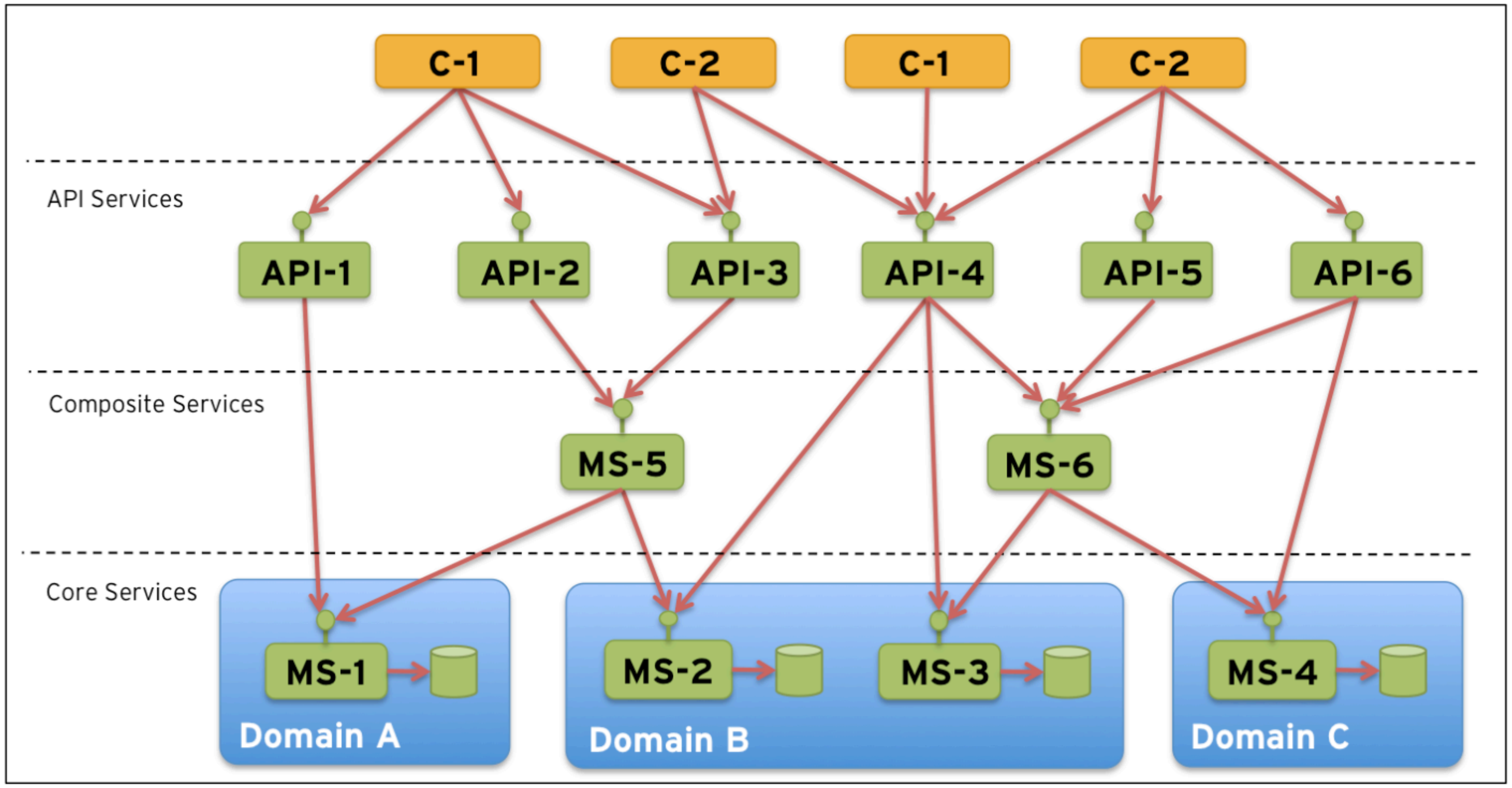
RECAP

- Microservices Architecture
  - Distributed Systems
  - Highly maintainable
  - Around Business Requirement
  - Reusable
- Spring Cloud
  - Eureka — Service Discovery
  - Feign — Http Client
  - Gateway — Unified Entry Point

ARCHITECTURE

- *Core services* Handling persistence of business data and applying business rules and other logic
- *Composite services* Composite services can either orchestrate a number of core services to perform a common task or aggregating information from a number of core services.
- *API services* Expose functionality externally allowing, for example, third parties to invent creative applications that use the underlying functionality in the system landscape.

…and horizontally we can apply some domain driven partitioning. This can result in a target architecture like:



SCENARIO

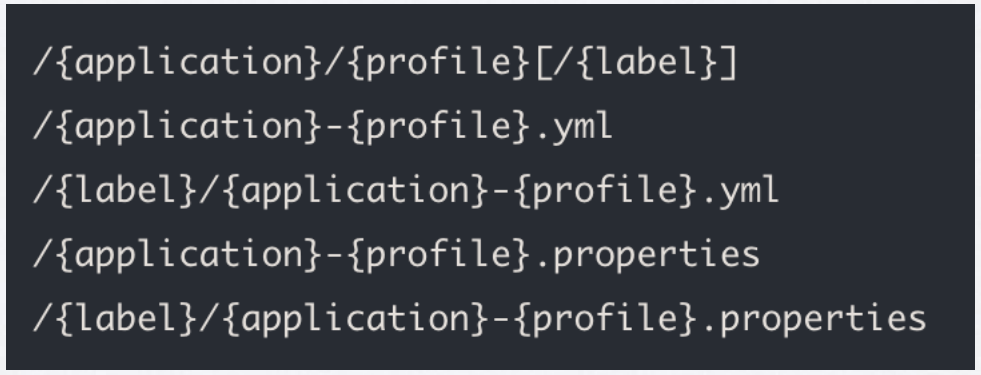
- When we configure our microservices, we have to add some properties
  - Register to Service Discovery
  - Gateway Configuration
  - Database connections (Some microservices may share same database)
  - Messaging Queue/Exchange
  - Redis Configuration
- Some of those configurations are same among all microservices
- What if we have 100 microservices?
  - D.R.Y. Principal

CLOUD CONFIGURATION

- Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system
- With the Config Server, you have a central place to manage external properties for applications across all environments.
- The concepts on both client and server map identically to the Spring Environment and PropertySource abstractions, so they fit very well with Spring applications but can be used with any application running in any language.
    - As an application moves through the deployment pipeline from dev to test and into production, you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate
- The default implementation of the server storage backend uses *git*, so it easily supports labelled versions of configuration environments as well as being accessible to a wide range of tooling for managing the content.

SPRING CLOUD CONFIG

- To get started, we have to
  - Add *spring-cloud-starter-config* to dependencies
  - Add *@EnableConfigServer* to the configuration
  - Add properties to connect to Git
- Query the configuration
  - The *Git*-backed configuration API provided by our server can be queried using the following paths



```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

  - where application = application name; profile = active profile; label = git branch

SPRING CLOUD CONFIG

- To use cloud configuration
  - Use @Value to get the properties from cloud configuration server
  - Add properties configuration using query method to fetch proper configuration
    - spring.application.name={application}
    - spring.profiles.active={profile}
    - spring.cloud.config.uri={configuration server host and port}
    - spring.cloud.config.label={label}
- Let's take a look at an example

SPRING CLOUD CONFIG

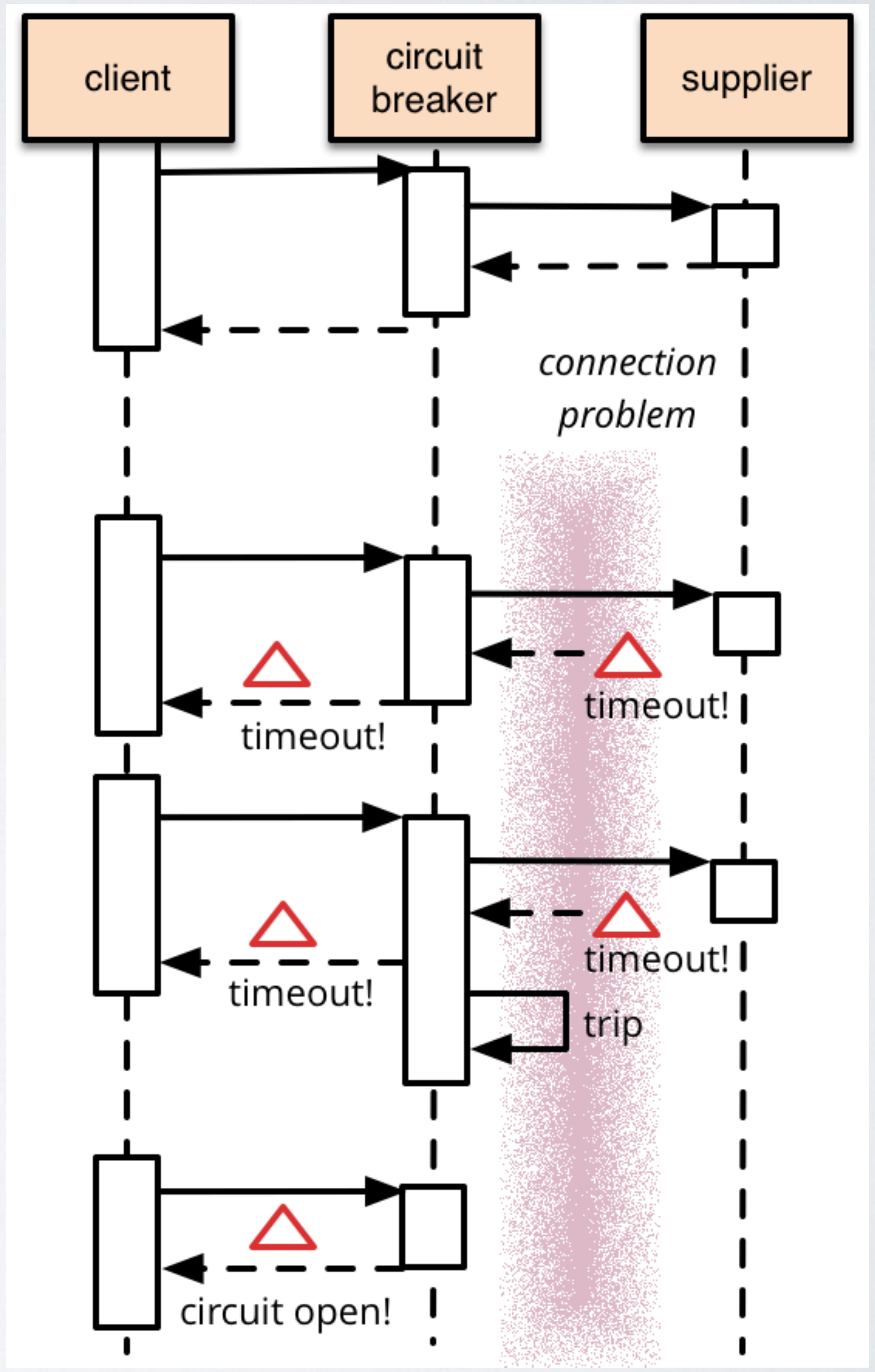
- Spring boot actuator provided us an endpoint "/refresh" which is used to refresh the cloud configuration.
    - In order to make it work, we have to use *@RefreshScope* on our configuration class, which will trigger a refresh event to update the properties
- However, it requires us to trigger the endpoint manually.
    - But we won't know if someone checks in or not
    - This process will becomes tedious if there are a lot of developers using the same cloud configuration server
- The better approach is to trigger the refresh event once there is any changes in Git and broadcast the event through all other available services

SCENARIO

- In a Microservices distributed system, we have following services (core services)
  - Account Service
  - Billing Service
- Now in order to process a request, we have to make a call to both services.
- What if one of the call failed?
- One of the big differences between in-memory calls and remote calls is that remote calls can fail, or hang without a response until some timeout limit is reached
- What's worse is that if there are multiple caller making calls to unresponsive supplier, it is very possible to run out of memory while waiting, which may lead to cascade failure in the application.

CIRCUIT BREAKER

- The circuit breaker pattern is the solution to this problem
- The basic idea behind the circuit breaker is very simple.
    - You wrap a protected function call in a circuit breaker object, which monitors for failures.
    - Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all



SPRING CLOUD NETFLIX HYSTRIX

- Spring Cloud Netflix Hystrix is an another module in Spring Cloud Project which implements the Circuit Breaker pattern
  - It is a fault-tolerance library, which is describing a strategy against failure cascading at different levels in an application
- The principle is analogous to electronics: *Hystrix* is watching methods for failing calls to related services.
  - If there is such a failure, it will open the circuit and forward the call to a fallback method.
  - The library will tolerate failures up to a threshold. Beyond that, it leaves the circuit open

SPRING CLOUD NETFLIX HYSTRIX

- To get started, we have to
  - Add spring-cloud-starter-hystrix to our dependency
  - Use *@EnableCircuitBreaker* on Configuration class
  - Create callback method
  - Configure the Feign and Hystrix threshold
- Let's take a look at an example

ADVANCE TOPIC

- Until now, we have covered all important part of microservices.
- However, when it comes to design a brand new distributed system like microservices, there are more to think about
  - CAP Principal (Consistency, Availability, Partition)
  - CI/CD (Continuous Integration / Continuous Delivery)

CAP PRINCIPAL

- Let’s use a example to understand the CAP
    - I am tired of IT job and wants to start a restaurant where I answer phone call to accept order and then delivery
    - So what I usually do is that I write down every order on a paper, gives it to kitchen and then DONE!
    - In the beginning, since my restaurant is new, there is few orders which I can handle all the phone calls. There is no overlap for different orders.

CAP PRINCIPAL

- After one month, my restaurant becomes more and more popular, so that I have to pick up phone calls from 10 am each day.
- However, no matter how hard I tried to answer all phone calls, there will be overlaps among phone calls, which lead to, one day, one customer walk to me and said "I have been trying to order my lunch for 30 minuet, but your phone is always busy!"
- I came up an idea that let me hire one more operator who can take the calls.
    - If one line is engaged, another person will pick up.
    - It took a week to onboard new person
    - This is improving *"Availability"*

CAP PRINCIPAL

- By hiring the new person to answer the phone call does help to reduce the wait time for ordering.
- I was very happy since there is no complaint until one day when I received phone call asking that what is my order status? After go through my order list, I don't find any order with this phone number. He definitely ordered with the other person!
- After talking to the new hire, I realized that it happened to him too.
- In order to resolve this problem, I came up with another great idea that we can exchange the order once received.
  - Once I received an order, I passed a copy to him; vice versa.
  - This is the *"Consistency"*

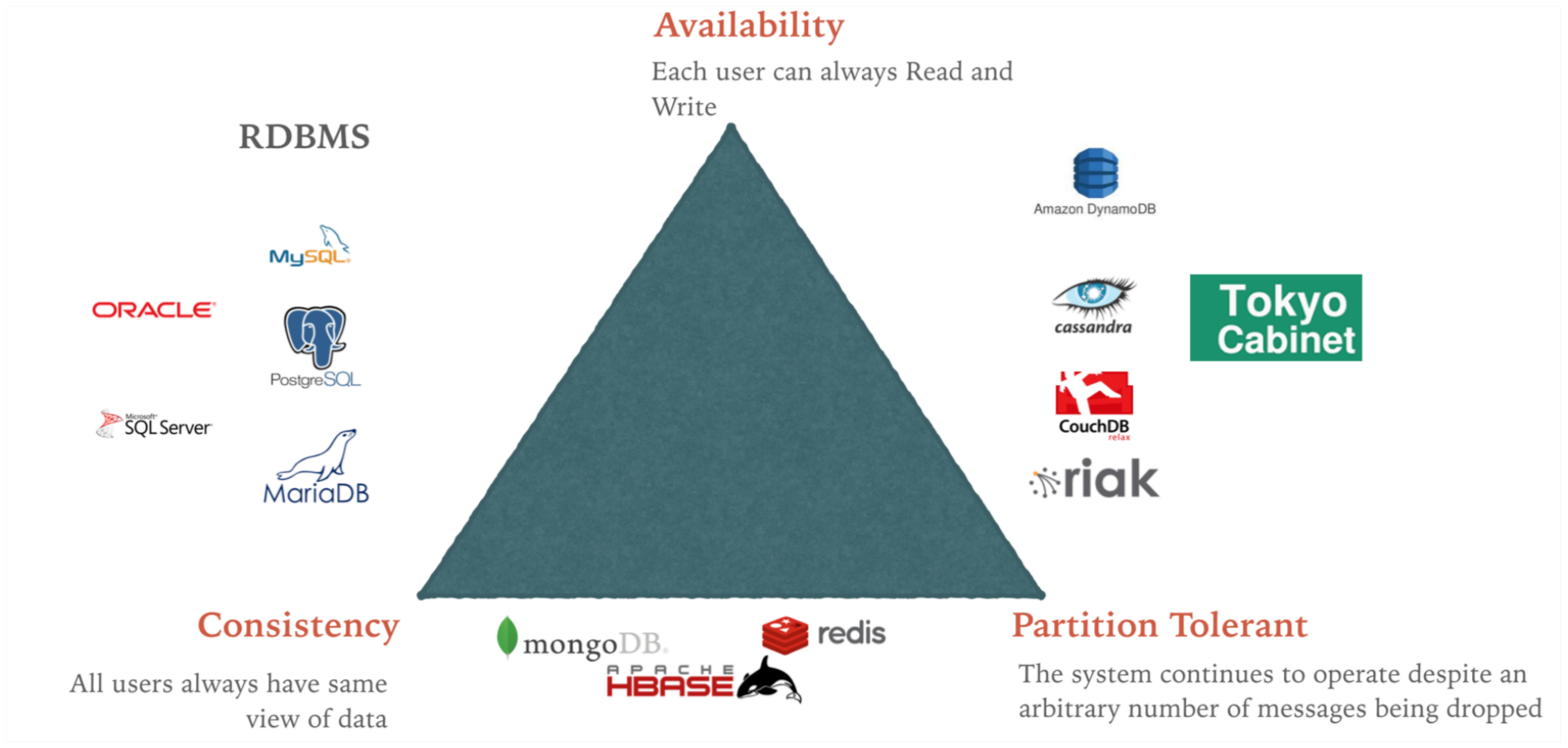
CAP PRINCIPAL

- This approach works well for a while.
- As my restaurant grow bigger, I hired two more persons to deal with the phone calls.
  - This improves the *"Availability"*
- However, sync up among four persons are really time consuming — we have to copy each order 3 additional times!
- What makes it worse is that the relationship between the 1st and 2nd person is pretty bad — They don't want to communicate with each other at all!

CAP PRINCIPAL

- CAP stands for Consistency, Availability and Partition Tolerance.
    - Consistency (C): All nodes see the same data at the same time. What you write you get to read.
    - Availability (A): A guarantee that every request receives a response about whether it was successful or failed. Whether you want to read or write you will get some response back.
    - Partition tolerance (P): The system continues to operate despite arbitrary message loss or failure of part of the system. Irrespective of communication cut down among the nodes, system still works.
- We can only achieve either PC or PA — Microservices are distributed/partitioned by nature!

CAP PRINCIPAL


Availability
Each user can always Read and Write
RDBMS
Consistency
All users always have same view of data
Partition Tolerant
The system continues to operate despite an arbitrary number of messages being dropped

CAP

ADVANCE TOPIC

- Centralize and Analysis logs: Elasticsearch, Kibana
- Container based deploy: Docker
- CI/CD — Continuous Integration/Continuous Delivery
  - Jenkins
  - Kubernetes

SUMMARY

- SQL
- Java SE
- J2EE
- Hibernate
- Spring Basics (Core, MVC, Data Access)
- Angular
- Spring Advance (Actuator, Security, Async, Repository, Caching)
- Spring Cloud (Microservices)
- Tools (SDLC, Git, Logging, JUnit, Server)

ANY QUESTIONS?