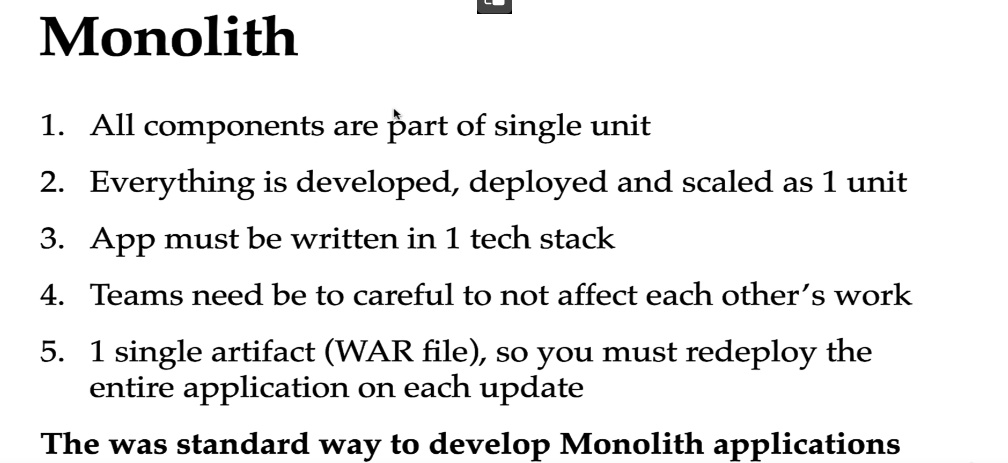
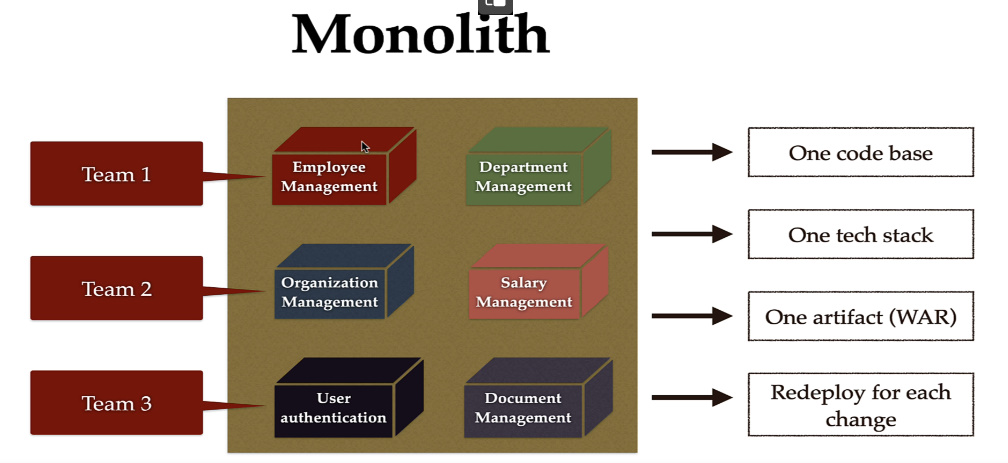
**Monolith**

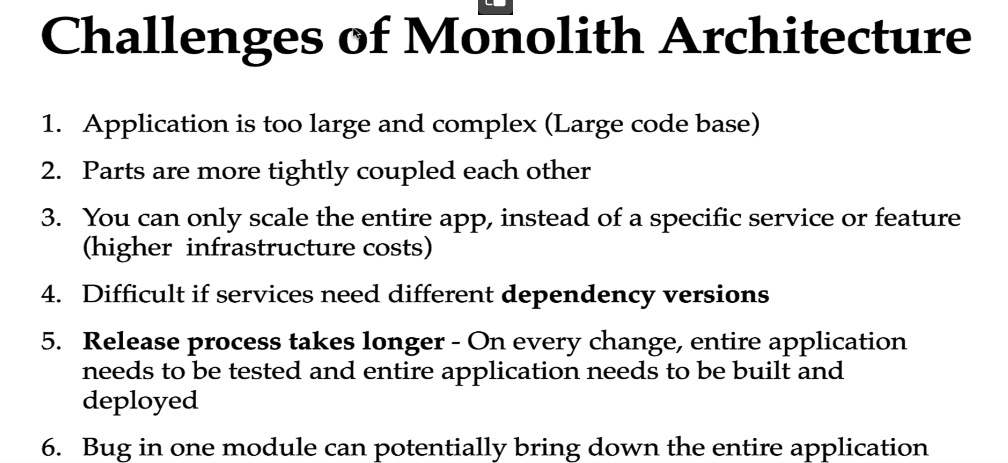


1. Totul codul aplicatiei e scris in acelas proiect, impreuna.
2. Totul e facut ca o unitate, ceea ce inseaman ca se scrie in acelasi limbaj de programare
3. Putem folosi doar o tehnologie, gen daca folosim Java, orice feature va fi in java, daca folosim maven, totul va integrat in el. Nu putem combina tehnologii
4. Toti lucreaza la acelasi lucru
5. Daca modificam ceva, trebuie iar sa facem un package cu tot proiectul si sa i dam redeploy



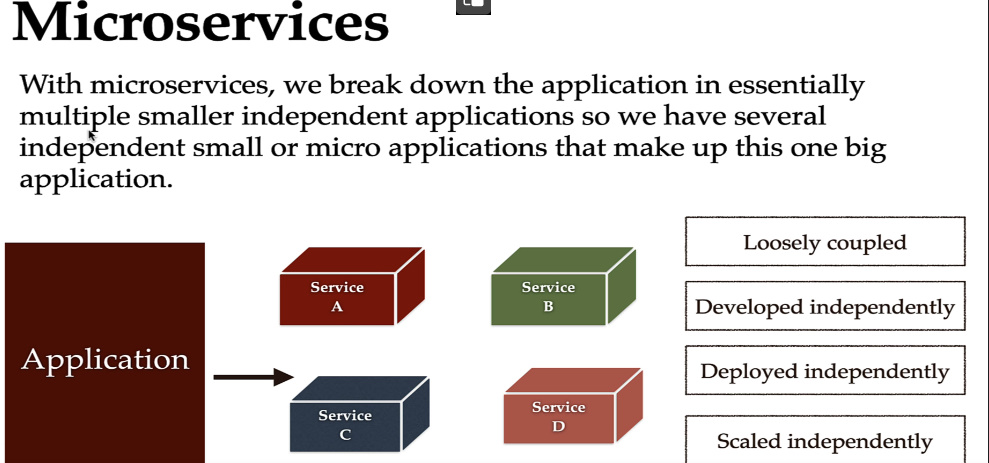
Aplicatia are difertie features, si toate sunt in acelasi proeict, toti lucreaza in acelasi project.

**Probleme cu monolith**

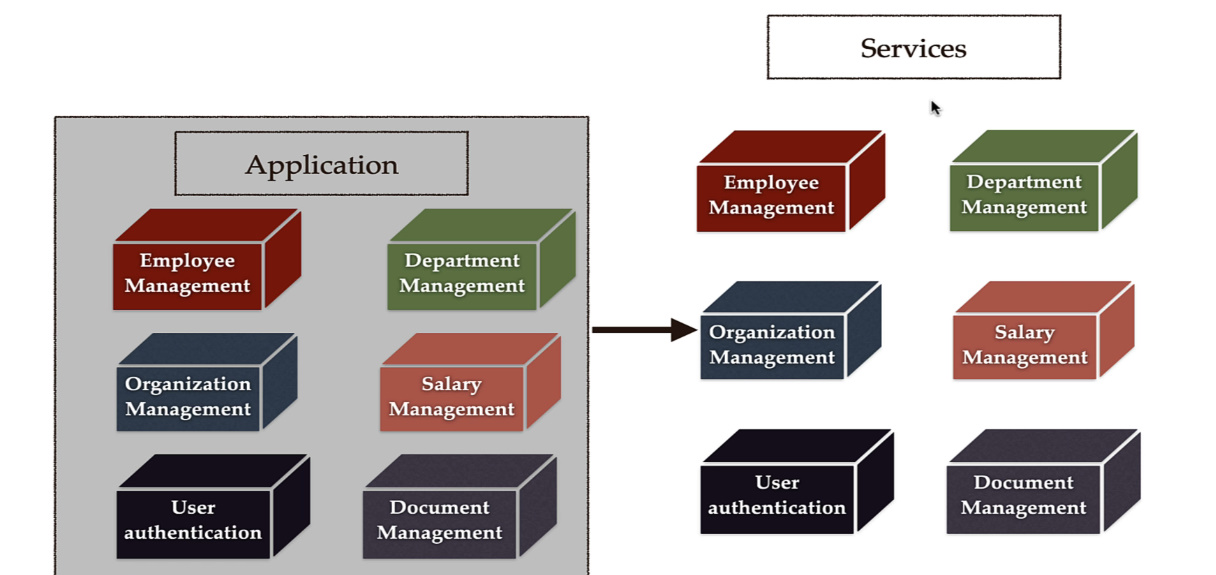


**Microservices**

* Microservices permite crearea de aplicatii care sunt compuse din servicii independente

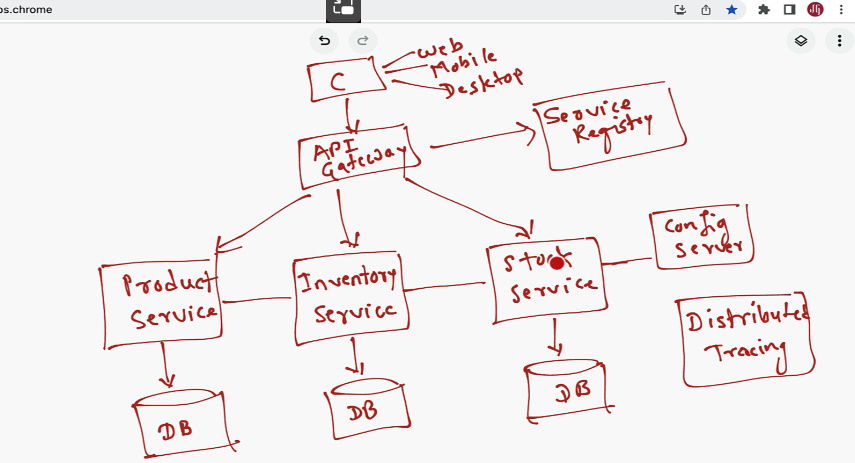


Deci in microservices, o aplicatie este rupta in bucati, in aplicatii mai mici, numite microservices





* De ex:



- **Fiecare Service are propria baza de date**

- Toate microserviciile sunt :

* **loosely coupled** – independente unele de altele
* **developed** independent
* **deployed** independent
* **scaled** independent – scale inseamna a servi mai multi useri.

- Serviciile comunica unele cu altele mereu

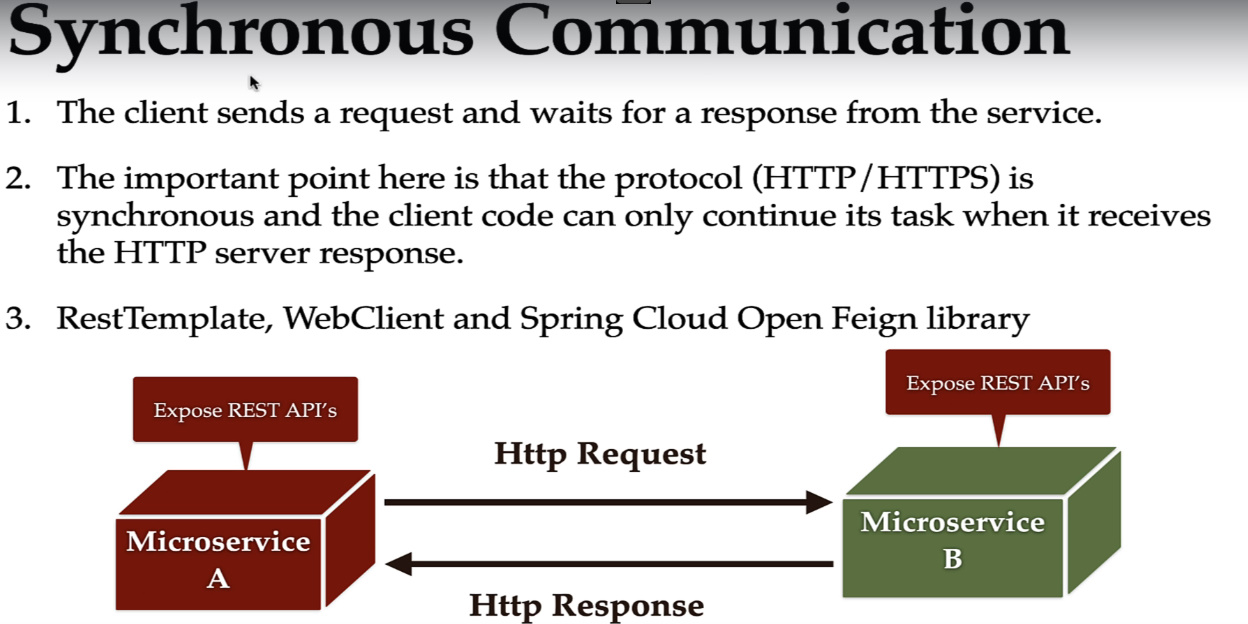
- Serviciile pot sa ofere cate un propriu REST API si fiecare service poate apela un REST API de la alt service

- Microserviciile au propriu lor code, si asa codul lor este mai mic si usor de inteles. Modificarile in aplicatie se fac asa mult mai usor

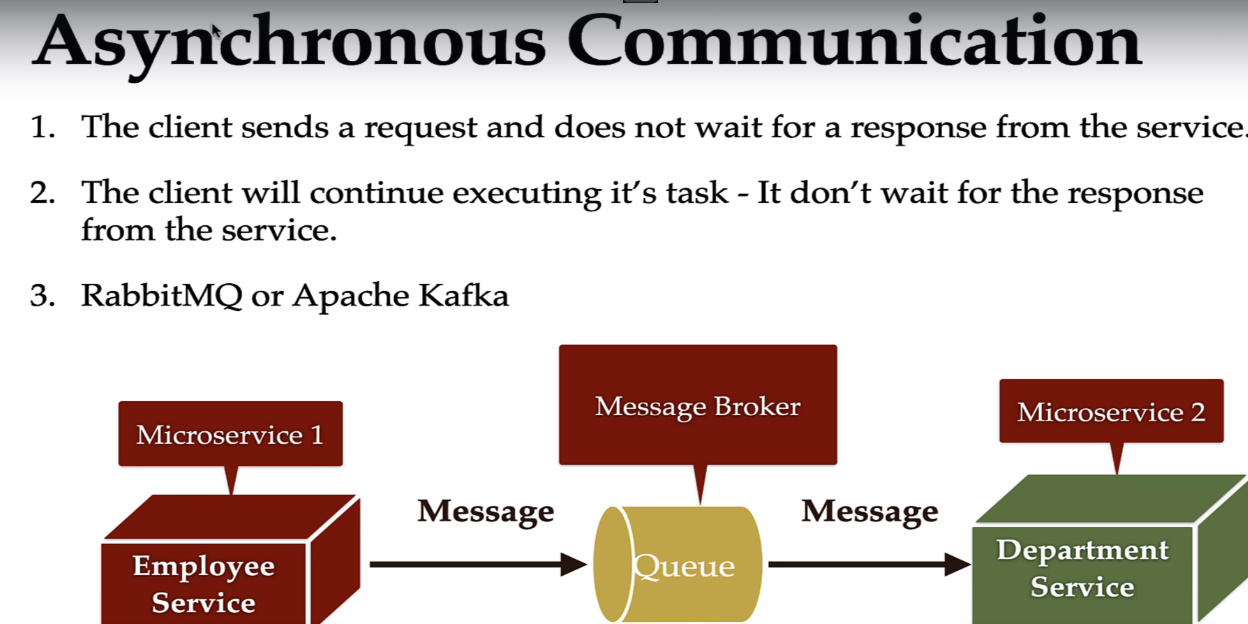
- Daca modificam doar un service, nu trebuie sa dam restart la intreaga aplicatie, ci doar la acel service

**Tipuri de comunicare intre microservices**

* **Synchronized** – comunicare prin HTTP Requests. Clientul trimite un request si asteapta un raspuns ca sa poata continua.

****

* **Asynchronized** – message brokers. Clientul trimite un request, dar nu asteapta niciun raspuns de la service.



**API Gateway pattern**

* API Gateway **pattern -** are grija ca fiecare request venit sa fie trimis catre servie corespunzator. Client ce il foloseste poate fi o webapp, desktop app etc. Cu el, client nu trebuie sa memoreze host si port la fiecare serive, dar va folosi niste reguli impuse de gateway, si gateway va gasi singur acel service.
* Deci, in loc de a folosi de ex path localhost:8081/api/employees la Employee microservies, ce are port de ex 8081, el va folosi port si host la gateway, de ex, care e 9191, asa localhost:9191/api/employees. Deci va folosi dar paths la diferite microservices, din rest controller a lor, si gateway singur va avea grija sa afle cui aparinte acest path, care e port si host la acel mciroservice si sa returneze raspunsul.

**Service Registery pattern**

* Toate datele despre microserviciile din proiect,in special numele lor, host si port vor fi inregistrate in **Service Registery**
* **El are urmatoarele roluri:**

**-** memoreaza si urmareste toate microserviciile inregistrate

**-** face ca serviciile sa fie constiente unele de altele

**-** ofera load balacing pentru microservicii

* Gateway,cand preia un request, foloseste Service Registery pentru a afla hostname si port folosind service registery apoi pe baza la host si port obtinut, apeleaza acel microservice.

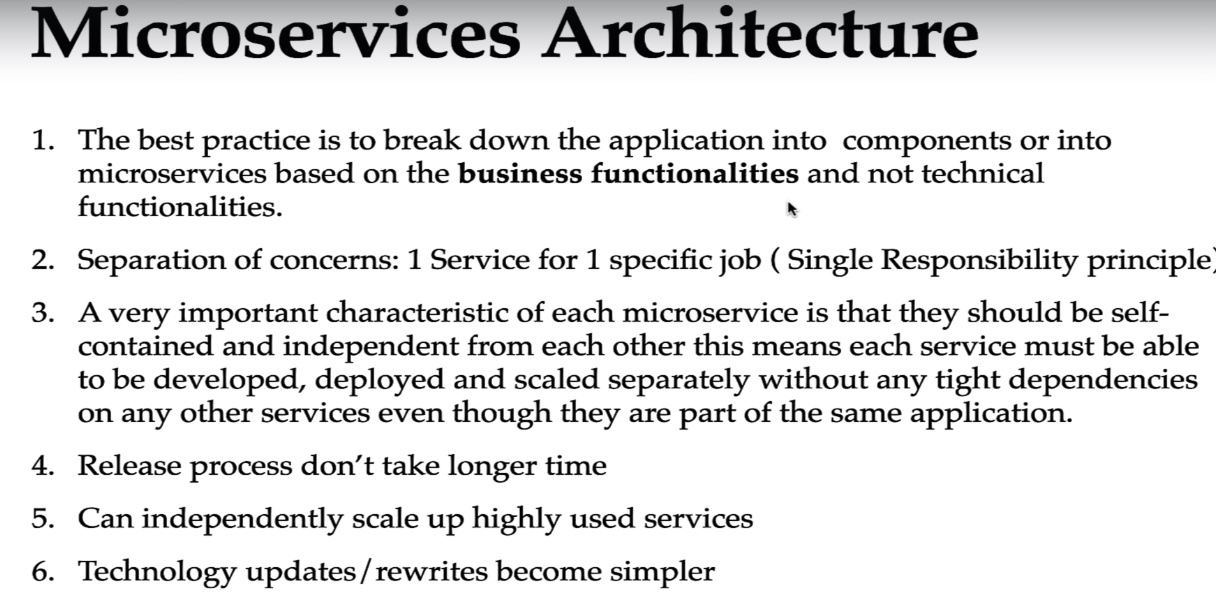
**Config Server pattern**

* **Config server pattern -**  externalizeaza configuratia microserviciilor. Fiecare microserviciu are un config file, si ca sa nu trebuiasca sa accesam fiecare mciroservice pentru a modifica config file a lui, config server le stocheaza pe toate si asa modificarea la mai multe config files se face din config server direct

**Distributed tracing pattern**

* **Distributed tracing** pastreaza loguri sau ierarhii de loguri pentru un HTTP call.
* **De ex,** fie ca avem mai multe microservices, si microservice1 il apeleaza pe microservice2. Daca gateway face apel la microservice1, microservice1 va apela microservice2 si avem nevoie poate la logam aceasta ierarhie de call, de la inceput la final

**Microservices Arhitecture**



- Deci, separarea in microservices se face dupa bussiness functionalities, nu dupa tehnical.

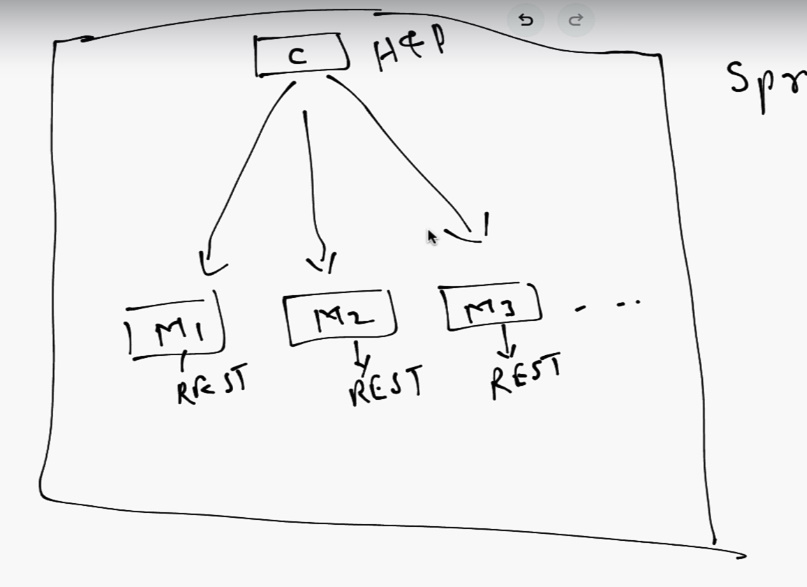
- Fiecare service trebuie sa aiba un job specific

- Un microservice nu depinde de altele! El trebuie sa poata fi dezvoltat independent!

- Fiecare microservice poate folosi propriile tehnologii si chiar limbaj de programare

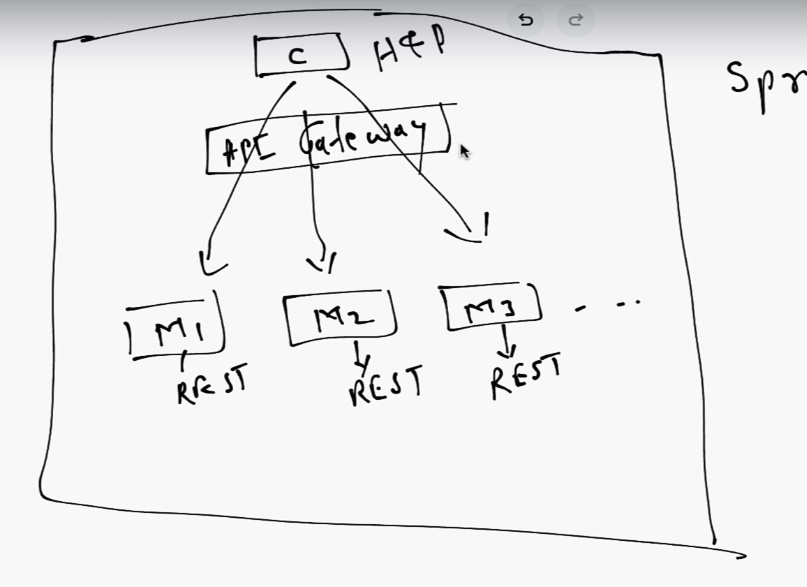
**Config file**

Fie un project cu 3 microservicii



Cu 3 microservicii, totul e usor. Fiecare microservice are un host si port si clientul trebuie doar sa foloseasca host:port ca sa apeleze un microservice. Dar, daca avem vreo 100 de microservici, atunci clientul nu are cum sa le tina minte pe toate, si sa le notam undeva si apoi sa le cautam va consuma foarte mult timp.

* Dar, putem cream un API Gateway care va prelua un request de la user si vagasi microservice necesar pentru a trimite acel request si response(daca e necesar), pe baza la Service Registry





**Fiecare microservice are un config file (CF) unde pastreaza configurarile sale.**

* Totusi, daca apare necesitatea de modifica config files de la mai multe mcroservicii, trebuie sa o facem manual in fiecare, si asta nu e deloc comod.
* **Config server** – pastreaza config file la mai multe microservices si permite modificarea lor direct din el,

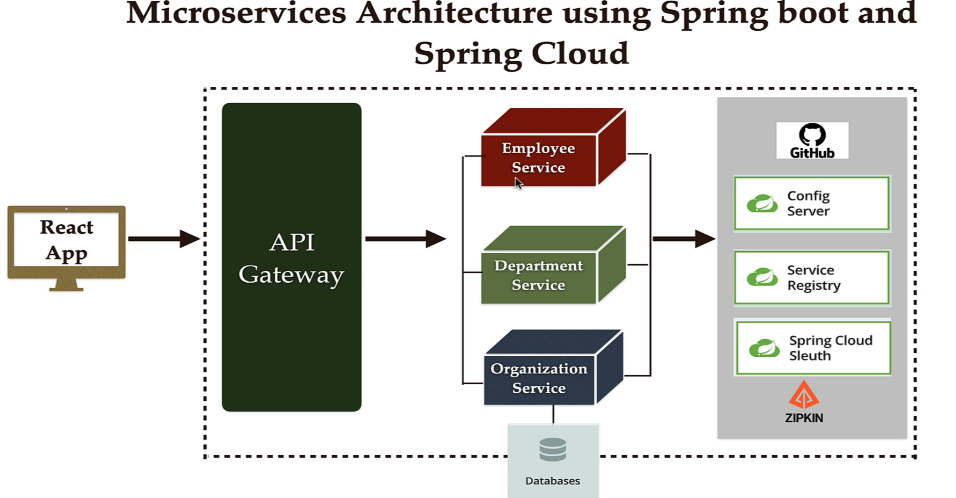
**Circuit breaker pattern**

* Fie un microservice1 si microservice2 si gateway. Client vrea sa apeleze Microservice, deci o va face prin gateway. Gateway va apela microservice1 si microservice1 sa zicem ca are nevoie sa apeleze microservice2, dar microservice2 e down, adica nu functioneaza
* In asa caz, microservice1 va tot face requesturi la microservice2 pana va fi returnata o eroare, ce va ajunge la gateway si apoi la client
* Faptul ca microservice1 apeleaza din nou microservice2 de multe ori e o pierdere de resurse si timp.
* **Circuit breaker** pattern limiteaza numarul de calluri pe care un microservice le poate face catre un alt microservice si stabileste reseponse in caz de error(cand un service e off si nu poate fi apelat de altul)

**Spring Cloud**

* Pentru a implementa toate patternurile necesare pentru a lucra cu microservices, ca api gateway, config server, circuit breaker, service registery , distributed tracing trebuie sa scriem cod manual.
* Dar, ca nu scriem code pentru toate, Spring Cloud deja le-a implementat.
* **Spring Cloud** – tool care implementeaza pattern necesare pentru a crea microservicii

**Ex de arhitectura**



Fiecare Microservice are baza lui de date

* **Server Registry si discovery pattern** – implementate de Spring Cloud Netflix Eureka Server
* **Config server** – va externaliza config file la fiecare microservice intr-un git repository. E implementat de Spring Cloud config module
* **API Gateway** – stabileste niste reguli pe baza la care va gasi service necesar, asa clientul nu va memora host si port la fiecare microervice. Implementat de Spring Cloud Gateway module
* **Distributed tracing** – implementat de **MicroMeter** module si Zipkin ca sa vedem asta intr-un user interface