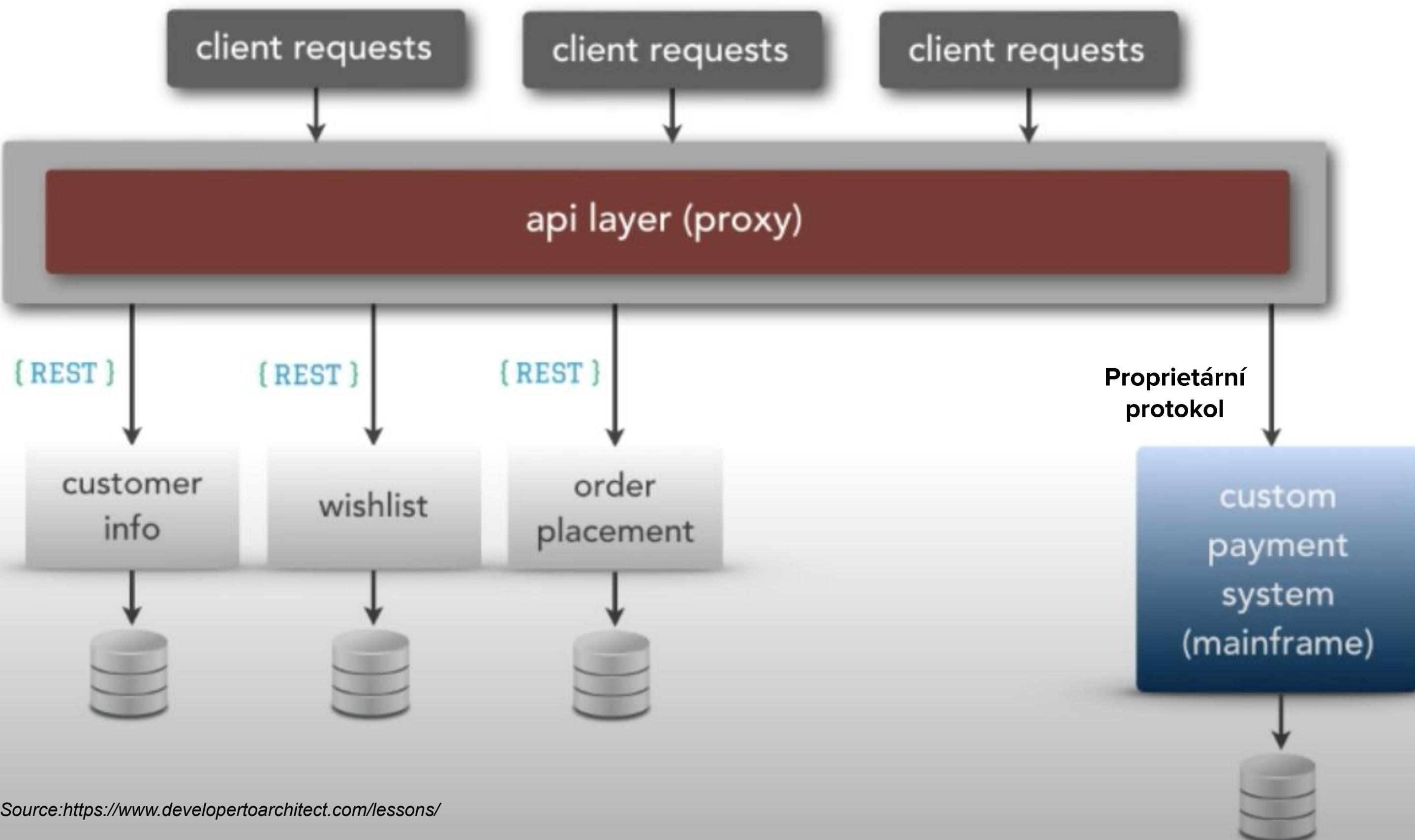
OMO

Microservisy 2 a úvod do reaktivních architektur

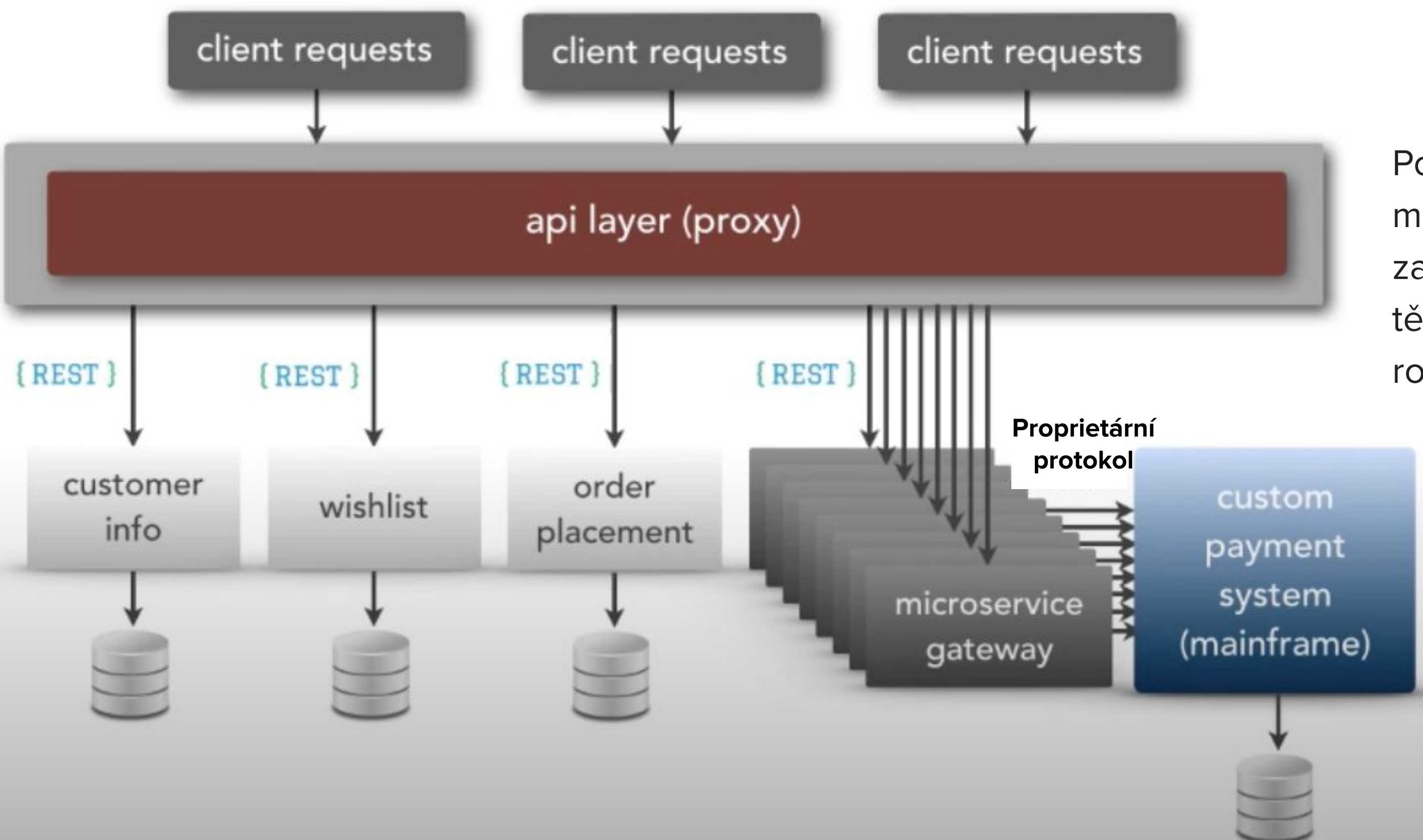
Ing. David Kadleček, PhD.
kadlecd@fel.cvut.cz, *david.kadlecek@cz.ibm.com*

Microservisní Gateway



Source:https://www.developertoarchitect.com/lessons/

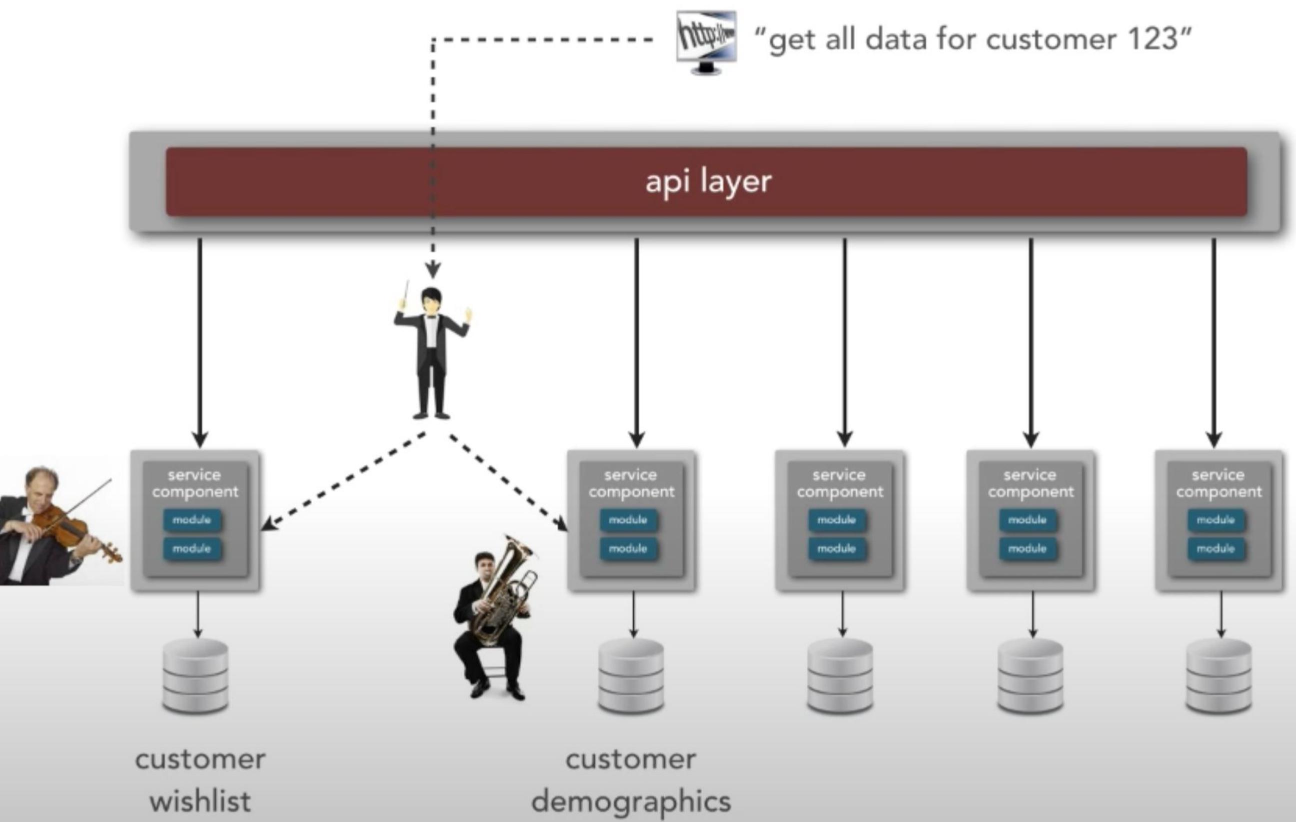
Microservisní Gateway



Potřebujeme do mikroservisní architektury zapojit starý systém s těžko integrovatelným rozhraním

Source:https://www.developertoarchitect.com/lessons/

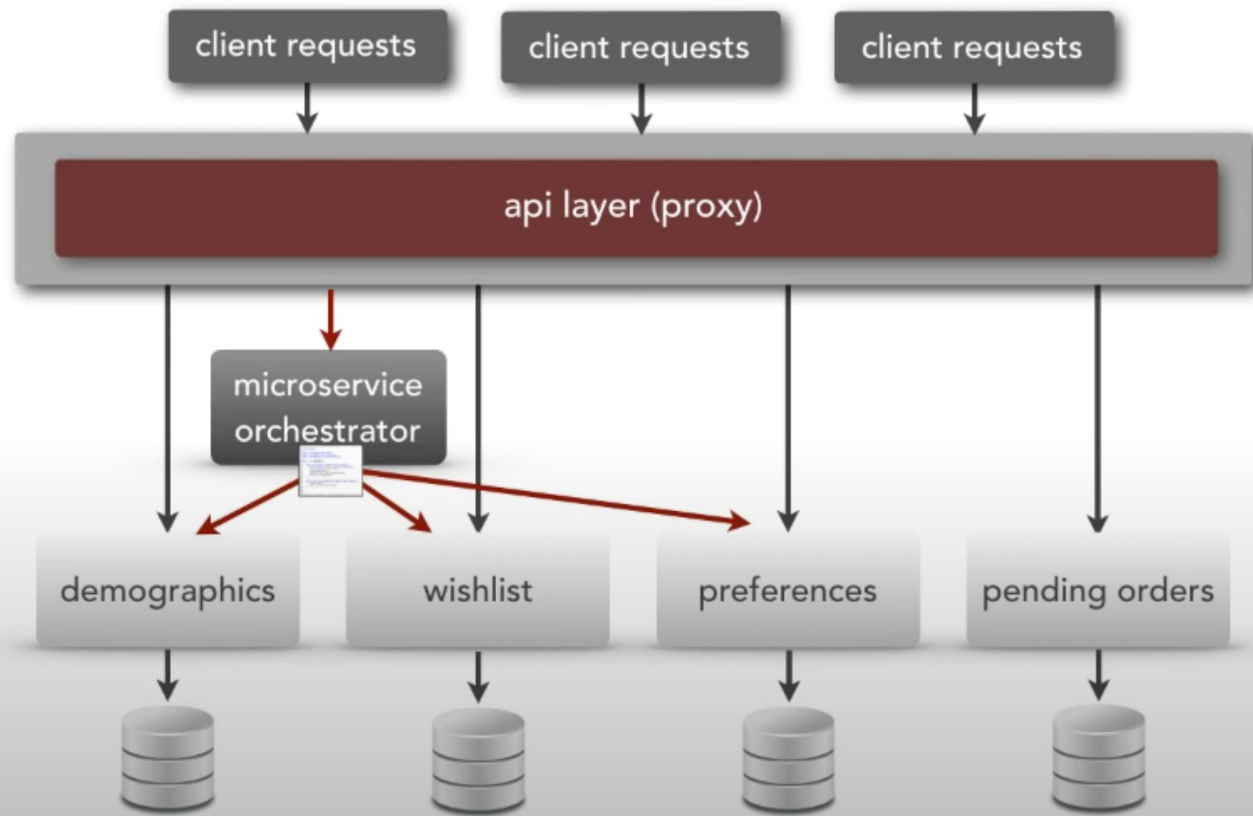
Orchestrace mikroservis



Potřebujeme službu, která vrací agregaci dat posbíraných z více služeb....

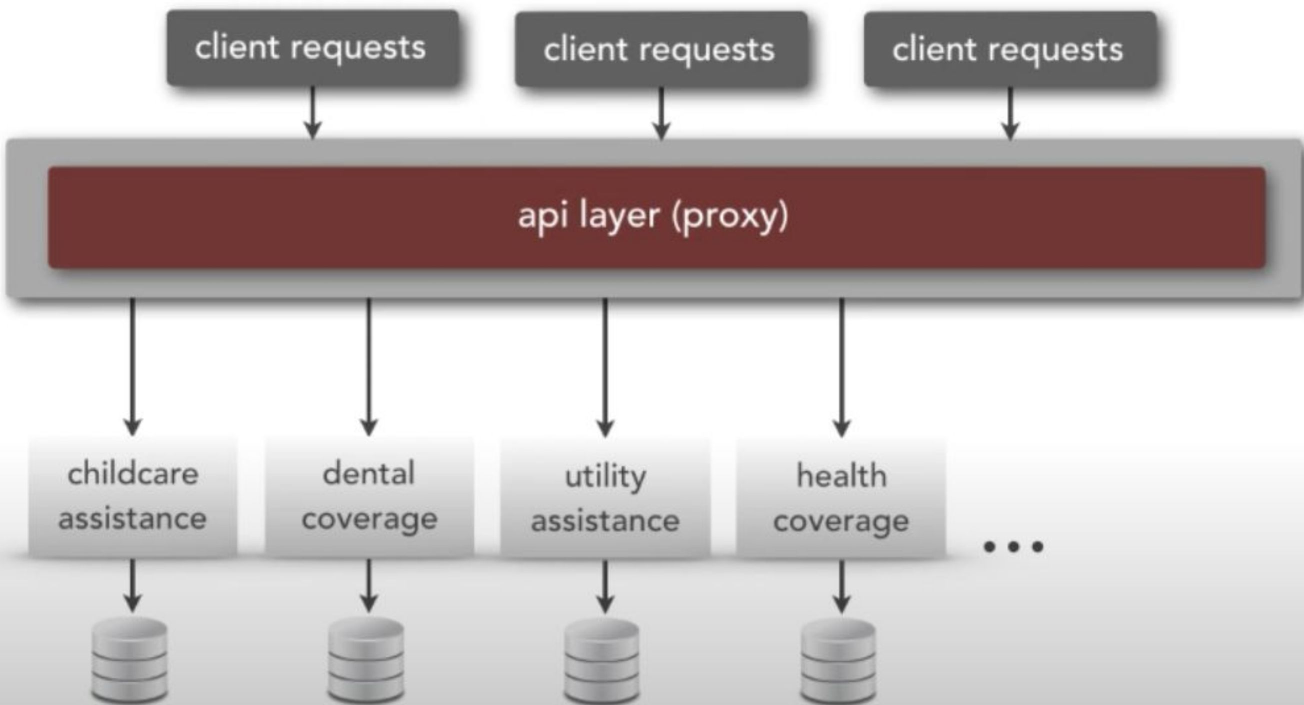
Source:https://www.developertoarchitect.com/lessons/

Orchestrace mikroservis



Source:https://www.developertoarchitect.com/lessons/

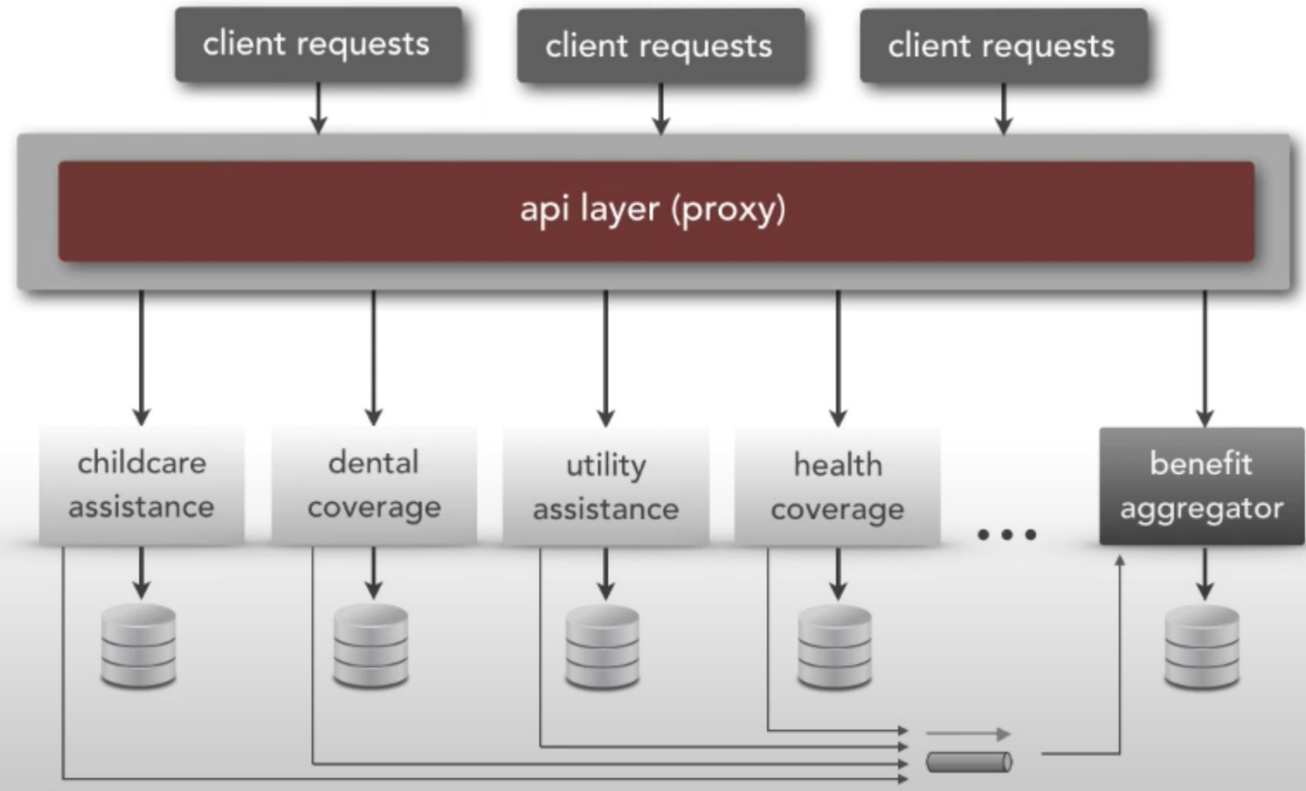
Agregace mikroservis



Potřebujeme službu, která nám vrací celkové benefity poskytované zákazníkovi. Za použití existujících služeb zákazníkem se mu přičítají benefitní body - vzniká tím úplně nová oblast dat.

Source:https://www.developertoarchitect.com/lessons/

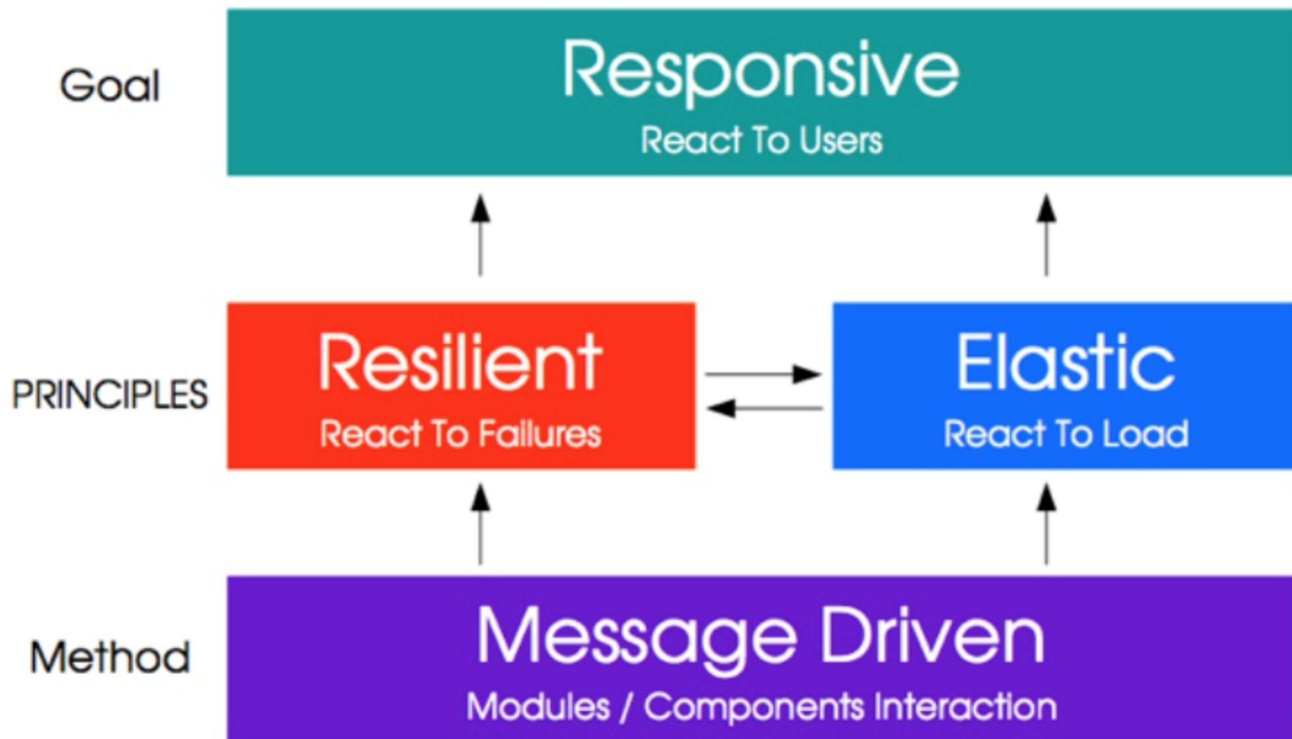
Agregace mikroservis



Source:https://www.developertoarchitect.com/lessons/

Reactive manifesto

Manifest reaktivních systémů definuje klíčové pilíře reaktivních architektur

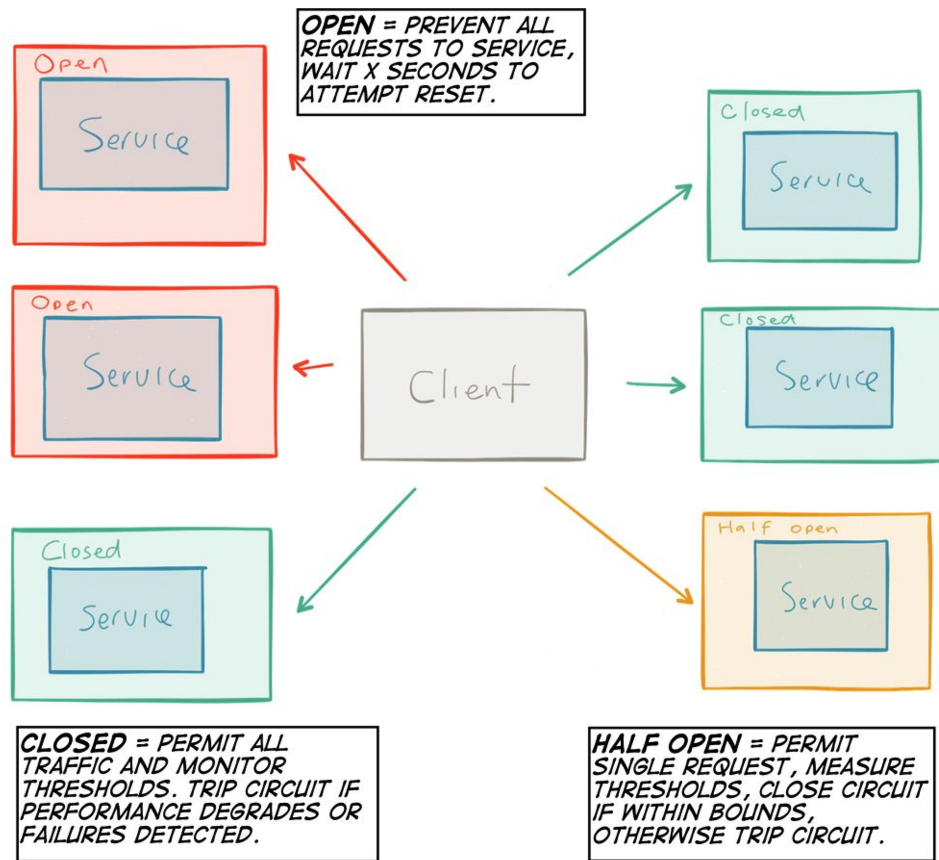


Circuit breaker

Circuit breaker = **jistič**

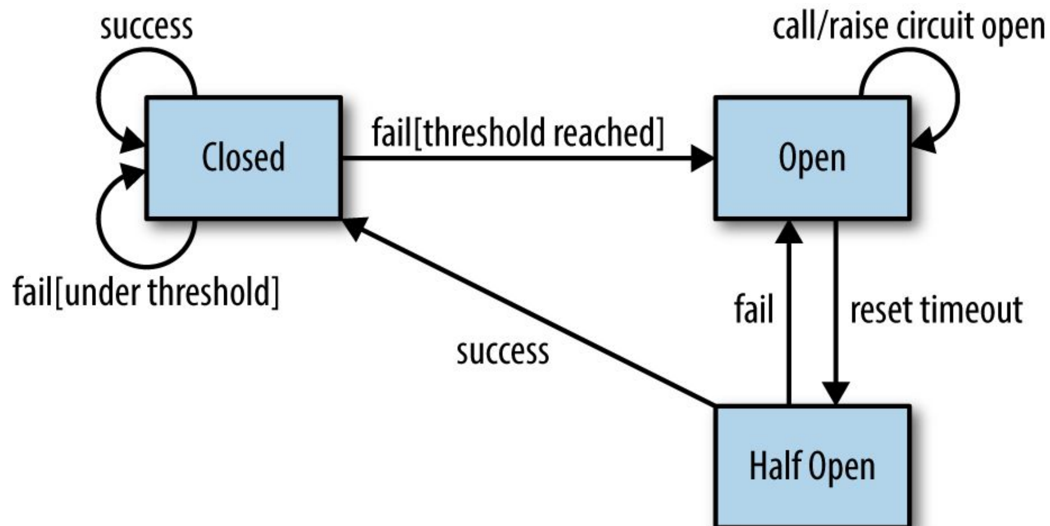
Wrapper obalující službu, který zajistí, že v případě, že je služba přetížená, tak na ni nechodí další požadavky a vrací uživateli okamžitou odpověď o nedostupnosti (uživatel nečeká až mu služba např. Až po minutě vrátí timeout)

Circuit Breaker pracuje pomocí měření odezev obalované služby nebo přímo instrumentací, kdy dostává monitoring data např. o utilizaci CPU a paměti komponenty ve které běží služba



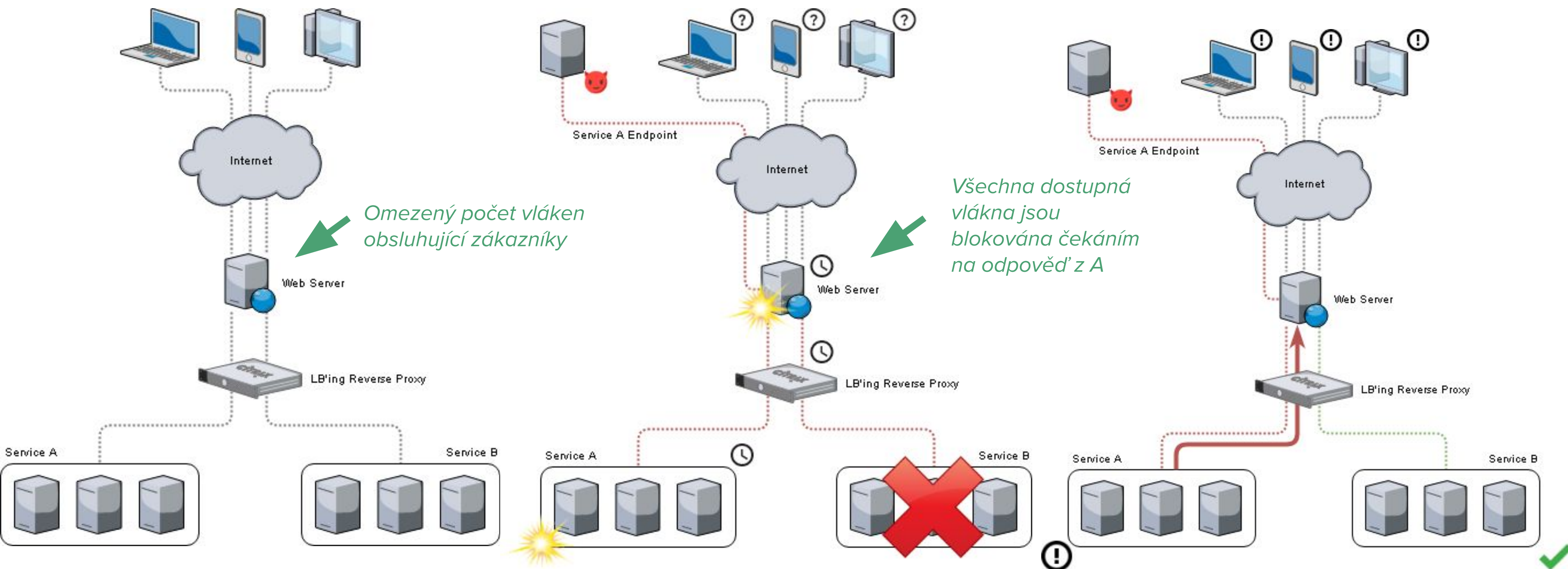
Circuit breaker

Speciální stav - napůl otevřený, kdy circuit breaker propouští jen omezený počet požadavků



Příkladem je např. Hystrix od Netflixu, který je embedovatelný ve Spring frameworku
Nebo Istio

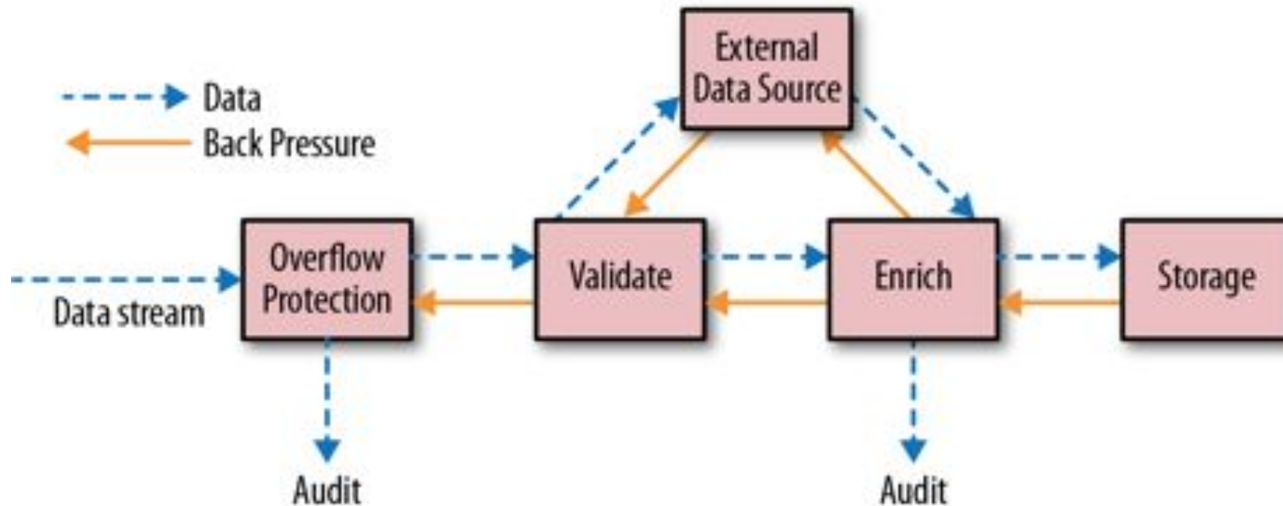
Scénář přetížení služby



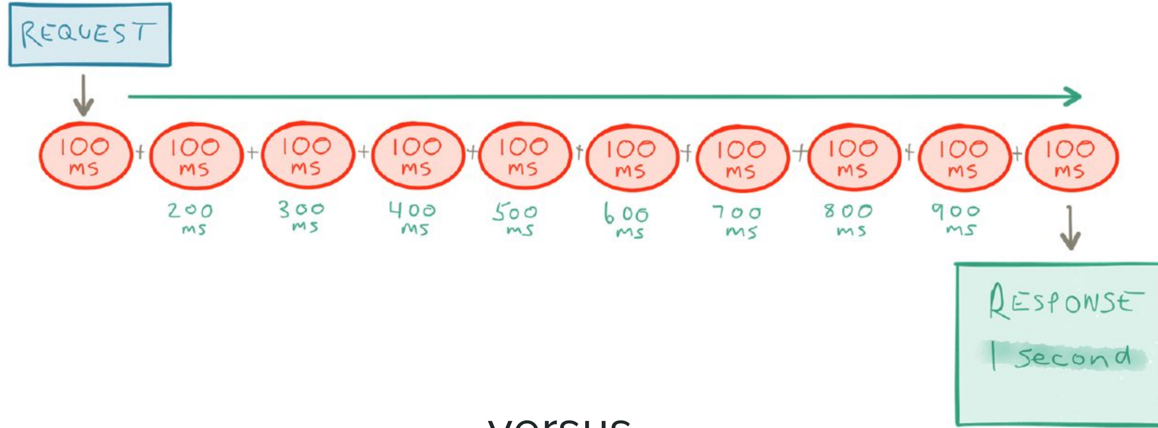
Back pressure

Mechanismus, kdy:

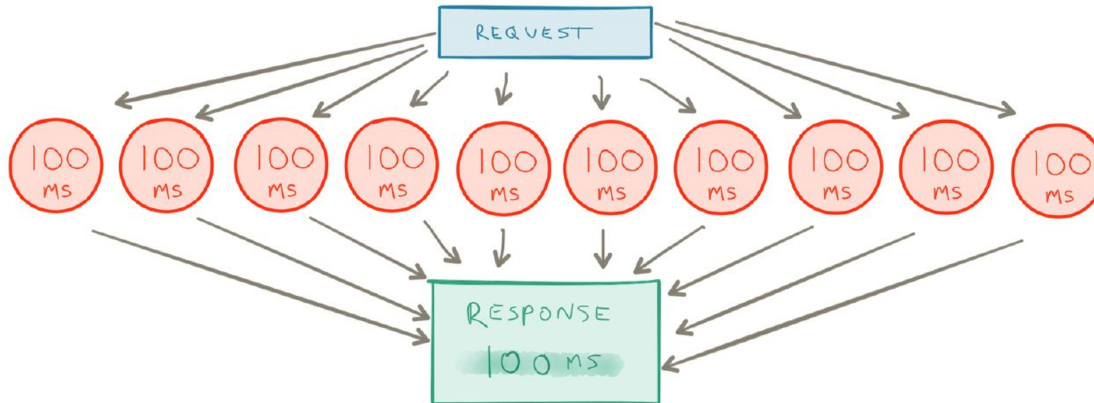
- postupně omezujeme load na službu, která nestíhá, abycho neblokovali další služby
- snažíme se load omezit mnohem blíže ke klientovi než jen těsně před službou, která je přetížená



Reactive manifesto cont.



versus



Future/Promise Pattern

Future je read-only view na proměnnou, která je nastavena někdy v budoucnu pomocí asynchronní funkce - ***Promise***. Před vlastním nastavením i po nastavení proměnné mohou vznikat různé situace, jejichž ošetření pattern také podporuje - jako např. timeout, vrácení chyby

Použití tohoto patternu výrazně snižuje blokování/latenci v distribuované nebo mnoha threadové aplikace.

Java realizace - Java realizuje tento design pattern pomocí frameworku okolo tříd *Future* a *CompletableFuture*. *Future* interface byl přidán do *Java 5* pro lepší podporu asynchronního zpracování. Nicméně až v *Java 8* byl přidán interface CompletableFuture, který navíc umožňuje výpočetní operace řetězit, zpracovávat různé druhy chyb (včetně timeoutu)

Design patterny Future/Promise a monáda jsou masivně využívány při psaní frontendových aplikací, kdy veškerá většina interakce mezi UI komponenty (model, view, controller), interakce se serverem je asynchronní.

Angular

```
  $scope.getRequest = function () {
          console.log("I've been pressed!");
          $http.get("http://urlforapi.com/get?name=Elliot")
              .then(function successCallback(response){
                  $scope.response = response;
              }, function errorCallback(response){
                  console.log("Unable to perform get request");
              }
      .catch(error => this.setState({ error, isLoading: false }));
```

React

```
fetch("http://urlforapi.com/get?name=Elliot")
      .then(response => {
        if (response.ok) {
          return response.json();
        } else {
          throw new Error('Something went wrong ...');
        }
      })
      .then(data => this.setState({ hits: data.hits, isLoading: false }))
      .catch(error => this.setState({ error, isLoading: false }));
```

Future/Promise Pattern

Metoda *complete()* - nastaví hodnotu proměnné, do té doby jsou blokovány všechny pokusy o získání její hodnoty

```
public Future<String> calculateAsync() throws InterruptedException {
   CompletableFuture<String> completableFuture = new CompletableFuture<>();
   Executors.newCachedThreadPool().submit(() -> {
       Thread.sleep(5000);
       completableFuture.complete("Hello");
       return null;
   });
   return completableFuture;
}
..
Future<String> completableFuture = calculateAsync();
String result = completableFuture.get(); //Program gets blocked at this line until result
provided
..
```

Zpracování chyby

V porovnání s try/catch blokem (syntaktický blok), v *CompletableFuture* se použije speciální metoda *handle()*. Jejími parametry jsou výsledek zpracování jestliže vše doběhlo OK a výjimka pokud nastal problém.

```
String name = null;
...
CompletableFuture<String> completableFuture
     = CompletableFuture.supplyAsync(() -> {
          if (name == null) {
               throw new RuntimeException("Computation error!");
          }
          return "Hello, " + name;
     })}).handle((s, t) -> s != null ? s : "Hello, Stranger!");

assertEquals("Hello, Stranger!", completableFuture.get());
```

Future a Promise Pattern

Runnable a Supplier rozhraní umožňují se dále zbavit kódu pro multi-threading pomocí metod *runAsync()* nebo *supplyAsync()* s pomocí lambda expressions následovně:

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello");
assertEquals("Hello", future.get());
```

Pokud chceme výsledek výpočtu dále zprocesovat funkcí, tak použijeme metodu *thenApply*, která vezme výsledek výpočtu, ten zprocesuje funkcí o vrátí CompletableFuture, který drží nový výsledek.

```
CompletableFuture<String> compFuture = CompletableFuture.supplyAsync(() -> "Hello");
CompletableFuture<String> future = compFuture.thenApply(s -> s + " World");
assertEquals("Hello World", future.get());
```

Jestliže nepotřebujeme vracet future bez návratové hodnoty - CompletableFuture<Void> future, tak se použije thenAccept()

```
CompletableFuture<Float> compFuture = CompletableFuture.supplyAsync(() -> 120.0);
CompletableFuture<Void> future = compFuture.thenAccept(s ->
ShoppingList.setPrice(s));
future.get();
```

Design pattern **Monáda** = kombinování futures

Zpracováváme více futures najednou, skládáme/vyhodnocujeme je sekvenčně, paralelně nebo dokonce hybridně.

Sekvenční zřetězení více futures

1) *thenCompose()* - výsledek zpracování funkce dáváme jako vstup do následující:

```
 CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> "Hello")
     .thenCompose(s -> CompletableFuture.supplyAsync(() -> s + " World"));
  assertEquals("Hello World", completableFuture.get());
```

Pozn. Java Stream map() je realizována kombinací thenCompose a thenApply
Rozdíl mezi thenCompose() a thenApply() je jako mezi Java Stream map() a flatMap(). Prosím prostudujte.

2) *thenCombine()* - výsledek dvou funkcí zkombinujeme do následující:

```
CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> "Hello")
      .thenCombine(CompletableFuture.supplyAsync(() -> " World"), (s1, s2) -> s1 + s2));
assertEquals("Hello World", completableFuture.get());
```

Design pattern **Monáda** = kombinování futures

Paralelní zřetězení více futures

1) ***allOf()*** - blokuje se dokud nejsou vyhodnoceny všechny výsledky:

```
CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> "Hello");
CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> "Beautiful");
CompletableFuture<String> future3 = CompletableFuture.supplyAsync(() -> "World");
CompletableFuture<Void> combinedFuture = CompletableFuture.allOf(future1, future2, future3);
combinedFuture.get();
assertTrue(future1.isDone());
assertTrue(future2.isDone());
assertTrue(future3.isDone());
```

2) ***anyOf()*** - blokuje se dokud není vyhodnocen jeden z výsledků:

```
CompletableFuture<String> future0 = createCFLongTime(0);
CompletableFuture<String> future1 = createCF(1);
CompletableFuture<String> future2 = createCFLongTime(2);
CompletableFuture<Object> future = CompletableFuture.anyOf(future0, future1, future2);
System.out.println("Future result>> " + future.get());
System.out.println("All combined>> " + future0.get() + "|" + future1.get() + "|" +
future2.get());}
```

Další příklad na **allOf()**

```
StringBuilder result = new StringBuilder();
List<String> messages = Arrays.asList("a", "b", "c");

List<CompletableFuture<String>> futures = messages.stream()
      .map(msg -> CompletableFuture.completedFuture(msg).thenApply(s -> delayedUpperCase(s)))
      .collect(Collectors.toList());
CompletableFuture.allOf(futures.toArray(new CompletableFuture[futures.size()])).whenComplete((v, 
th) -> {
      futures.forEach(cf -> assertTrue(isUpperCase(cf.getNow(null))));
      result.append("done");
});
```