

Архитектурни стилове на софтуерни системи (част 2) -Резюме

- Architectural patterns in cloud
- Service-oriented architectures: SOAP services, Microservices, REST services

Architectural Styles in Cloud (Архитектурни стилове в облака)

Circuit Breaker (Циркуйт Брейкър)

Неуспех на услугите в разпределена среда - Класическо решение е да се имплементира **timeout** за др. услуги, които я извикват. Води до ненужно използване на ресурси в облачна среда. Пр. стотици потребители, чакащи за отказала услуга, всеки с изчакване -**timeout**.

Circuit Breaker Pattern  (Патърн „Прекъсвач на веригата“):

-Предотвратява да се извършват операции, които е **малко вероятно да успеят**- извикване услуга, вече известна като неуспешна.

-Действа като **proxy** за операции, които могат да се провалят.

-Наблюдава **броя на последните неуспехи** и решава дали да позволи операцията или да върне изключение веднага.

Предимства:  Представи си, че ел. уред в дома ти създава късо съединение. Ако нямаш предпазител (circuit breaker), целият ток в къщата може да спре. Но ако имаш предпазител, той прекъсва само проблемната част, предпазвайки останалите уреди. 

- Повишава надеждността на системата.
- Приложенията **няма да чакат излишно за timeout**, когато услугата е известна като неработеща.
- Полезен при **временни прекъсвания на мрежата** или когато услугата е твърде **натоварена**.

Sharding Style (Стил „Шардинг“)  - Целта е увеличаване на мащабируемостта при работа с големи обеми от данни. Хранилището се разделя на данни - хоризонтални партиции, наречени **shards**. Техника за разделяне на големи обеми от данни на по-малки, по-лесно управляеми части-shards. Вместо цялата информация да се съхранява в една огромна база данни, тя се разпределя между няколко отделни сървъра или бази.

Мотивация за използване: Увеличаване капацитета за съхранение; Повишаване на изчислителните ресурси; Оптимизация на мрежовата пропускателност; Географско разпределение на данните.

Ключови фактори: Важно е как ще бъдат разделени данните между shards. **Shard key (ключ за разделяне)** определя как се разпределят данните- статичен и не зависи от данни, които може да се променят.

Имплементация: Логиката за sharding може да се реализира в приложението или да е част от самата база данни. Стратегии за импл:

-**Lookup strategy**  (Стратегия за търсене)- **списък с правила или таблица**, която казва къде точно се намират данните. Вместо да изчисляваш местоположението на данните, просто проверяваш в „карта“ (lookup table), за да видиш в кой shard се намират. Имаш потребители от различни държави: **Shard 1:** Потребители от България; **Shard 2:** Потребители от Германия; Когато някой потребител влезе, системата проверява в таблица: „Потребителят е от България → изпрати заявката към Shard 1.“

-**Range strategy** (Стратегия с диапазони)- данните се разделят на диапазони (ranges)- определени стойности се съхраняват в определени shards **Shard 1:** ID от 1 до 250,000, **Shard 2:** ID от 250,001 до 500,000

-**Hash strategy**  (Хешираща стратегия)- използва **хешираща функция** (hash function), за да определи в кой shard да се запишат

или прочетат данните. Хеширането взема стойност (потребителско ID) и я преобразува в друг числов резултат, който определя shard-а. Имаш 4 shard-а и искаш да запишеш потребител с ID 12345. Системата изчислява: $\text{hash}(12345) \% 4 = 1$ - данните отиват в **Shard1**.

Предимства: По-добро управление на данните; Повищена производителност на хранилището.

Недостатъци:

- Сложност при определяне на местоположението на данните.
- Повече редици разходи, когато данни от една заявка са разпръснати в много shards.
- Неравномерно разпределение на данните.
- Трудности при проектиране на универсален shard key.

Queue (Опашка)

Проблем: В облака услуги могат да бъдат претоварени от множество едновременни заявки. **Решение:** Използване на queue за балансиране на натоварването и избягване на отказ на услуги. **Видове опашки:**

-Standard Queue (Стандартна опашка) - Действа като буфер за съхранение на съобщенията, докато услугата ги обработи и особено когато има голям обем от заявки. Минимизира риска от недостъпност при голям брой заявки. Няма приоритизация –чака наравно с всички останали.

-Priority Queue (Приоритетна опашка) - Сортира заявките според приоритет/важност/. Приоритетът се задава от приложението или автоматично от опашката. По-важните заявки се обработват първи, независимо кога са постъпили. Позволява бързо реагиране на критични заявки. Подобрява производителността за важни операции

- **Fixed Length Queue (Опашка с фиксирана дължина)** - Предотвратява Denial of Service (DoS) атаки чрез ограничаване

на броя на съобщенията. Когато лимитът бъде достигнат, се връща изключение вместо изчакване на timeout.

Cache (Кеш) - Използва се за оптимизиране на многократния достъп до често използвани данни. **Процес:**

Четене: Проверка в кеша. Ако липсва, данните се извличат от хранилището и се съхраняват в кеша.

Запис: Промяна на данните в хранилището. Инвалидиране на съответния елемент в кеша.

Service Oriented Architectures - Сервизно-ориентирани

Software Services (Софтуерни услуги) - софтуерен компонент, който може да бъде достъпен от отдалечени компютри през интернет. При подаден вход, услугата генерира съответен изход без странични ефекти.

-Услугата се достъпва чрез нейния публикуван **interface (интерфейс)** и всички детайли на имплементацията ѝ са скрити.

-Услугите не поддържат вътрешно състояние. **State information (информация за състоянието)** се съхранява в база данни или се управлява от заявителя на услугата.

-При заявка към услуга, информацията за състоянието може да бъде включена като част от заявката, а актуализираното състояние се връща като част от резултата.

-Поради липсата на локално състояние, услугите могат динамично да се преразпределят между виртуални сървъри и да се репликират на няколко сървъра.

Web Services (Уеб услуги)- технологии, които позволяват различни приложения да комуникират помежду си през интернет. Използват стандарти и протоколи, които улесняват взаимодействието между

разл. С и платформи. Уеб услугите се изграждат върху концепцията за **сервисно-ориентирано изчисление** (service-oriented computing) С се състоят от независими, но взаимодействащи помежду си услуги.

През 90-те години, след различни експерименти със сервисно-ориентирано изчисление, концепцията за ‘big’ Web Services започва да се оформя в началото на 2000-те. Тези услуги се основават на **XML** (Extensible Markup Language), за да осигурят **формати за данни**, които могат да бъдат обработвани от разл. платформи, като: **SOAP** (Simple Object Access Protocol) за взаимодействие м/у услуги; **WSDL** (Web Services Description Language) за описание на интерфейса; **UDDI** - Universal Description, Discovery and Integration - стандарт, който позволява **описване, откриване и интеграция на уеб услуги**. С за регистриране на уеб услуги, така че други приложения или услуги да могат да ги намерят и използват.

Представи си, че имаш уеб сайт за резервации на хотели. Този сайт трябва да се свърже с различни доставчици на услуги, като: Услуги за обработка на плащания; Услуги за проверка на наличността на хотели; Услуги за потвърждения на резервации. Всяка от тези услуги може да бъде реализирана като **уеб услуга, която комуникира чрез SOAP, е описана чрез WSDL и може да бъде открита и интегрирана чрез UDDI**.

Повечето софтуерни услуги не изискват сложността, присъща на уеб протоколите, затова съвременните С използват по-прости и ефективни методи с по-малък **overhead** (режийни разходи) - „леко тежки“ (**lightweight**) протоколи за взаимодействие, които имат по-ниски разходи и по-бързо изпълнение.

Описание на уеб услуга (Web Service Description Language - **WSDL**) - Интерфейсът на услугата се дефинира в описание на услугата, изразено в WSDL. **WSDL спецификацията определя:**

- Какви операции поддържа услугата и формата на съобщенията, които се изпращат и получават.

- Как се достъпва услугата - свързването (binding) картографира абстрактния интерфейс към конкретен набор от протоколи.
- Къде се намира услугата- URI (Universal Resource Identifier).

Микроуслуги (Microservices)-малки, безсъстояние(stateless) услуги с една отговорност. Те се комбинират за създаване приложения.

- Те са **напълно независими със собствена база данни и код за управление на потребителския интерфейс (UI)**.
- Софтуерните продукти, използващи микроуслуги, имат **архитектура от тип микроуслуги** (microservices architecture).
- Ако е необходимо създаване на софтуерни продукти за **облака**, които са адаптивни, мащабируеми и устойчиви, препоръчително е да бъдат проектирани около архитектура от микроуслуги.

Мартин Фаулър обяснява, че когато се говори за архитектурни стилове като **монолит и микросервизи**, не трябва да се разглеждат като строго противоположни или взаимно изключващи се. Много С може да имат елементи и от двете- попадат в „размита зона“ м/у тях.

RESTful уеб услуги (RESTful Web Services) - Уеб услугите, базирани на XML стандарти, са критикувани като „**тежки**“- “over-general” и неефективни. REST (REpresentational State Transfer) е архитектурен стил, базиран на трансфер на представяния на ресурси от **сървър към клиент**. Този стил е в **основата на уеб** като цяло и е по-прост за реализиране уеб услуги в сравнение със SOAP/WSDL.

- RESTful услугите изискват по-ниски разходи и се използват от много организации, които изграждат системи, базирани на услуги.

Ресурс (Resource) - Основният елемент в RESTful архитектурата - елемент от данни като каталог, медицински запис или документ. Има множество представяния: MS Word, PDF, Quark Xpress, JSON. Операции върху ресурси: **Create, Read, Update, Delete**

HTTP REST URL: `https://api.example.com/users?id=123`

Протокол (Protocol) - http:// или https://

- Определя транспортната схема, която ще се използва за изпращане и получаване на заявката.
- HTTP (HyperText Transfer Protocol) е най-често използваният.
- HTTPS е защитена версия, която криптира данните.

Име на хост (Host Name) - api.example.com

- Идентифицира сървъра, на който се намира ресурсът.
- Това е домейн или IP адрес, който посочва точното местоположение на ресурса в интернет.

Път (Path) /users

- Определя конкретен ресурс на сървъра, който искаме да достъпим.
- Структурата на пътя може да показва йерархия от ресурси.

Стринг на заявка (Query String) ?id=123

- Използва се за филтриране или персонализиране на заявката.
- Започва със символа ? и съдържа двойки ключ-стойност (**ключ=стойност**), разделени със & при повече параметри.

Протокол: `https` – методът за комуникация; **Хост:** `api.example.com` – сървърът, където се намира ресурсът; **Път:** `/users` – кой ресурс искаме; **Стринг на заявка:** `?id=123` – параметър, който уточнява кой точно потребител искаме (с ID = 123)

Недостатъци на RESTful подхода: При сложен интерфейс може да е трудно да се проектира набор от RESTful услуги; Липсват стандарти за описание на RESTful интерфейси- налага използването на неофициална документация за разбиране на интерфейса.