MASARYKOVA
UNIVERZITA

FAKULTA INFORMATIKY

Integrace modulu distribuované cache v rámci platformy Kentico

Diplomová práce

BC. JIŘÍ MALEC

Brno, podzim 2023

MASARYKOVA UNIVERZITA

FAKULTA INFORMATIKY

Integrace modulu distribuované cache v rámci platformy Kentico

Diplomová práce

BC. JIŘÍ MALEC

Vedoucí práce: prof. RNDr. Tomáš Pitner, Ph.D.

Softwarové inženýrství

Brno, podzim 2023

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Bc. Jiří Malec

Vedoucí práce: prof. RNDr. Tomáš Pitner, Ph.D.

Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce panu prof. RNDr. Tomáši Pitnerovi, Ph.D. za cenné rady, jenž mi poskytl během psaní, za jeho ochotu a čas, který mi věnoval. Dále děkuji za možnost psaní práce ve spolupráci se společností BiQ Bluesoft a.s. Na závěr děkuji rodičům za podporu po celou dobu studia.

Shrnutí

Cílem diplomové práce je navrhnout a implementovat vlastní modul umožňující integraci technologie distribuované cache paměti do produktu Kentico Xperience. V první části je představen koncept webového cachování a jsou porovnány přístupy lokální a distribuované cache. V následující části představuji způsoby, jak jsou tyto přístupy realizovatelné v prostředí ASP.NET. Dále popisuji existující mechanismy cachování v prostředí Kentico Xperience. Na základě stanovených požadavků na vznikající modul je provedena analýza a srovnání tří systémů distribuované cache: Memcached, Redis a Hazelcast. V závěrečné části práce provádím čtenáře procesem návrhu a implementace vlastního modulu.

Klíčová slova

Webová cache, Distribuovaná cache, Kentico Xperience, Redis, Hazelcast, Memcached, ASP.NET

Obsah

Úvod **1**

1 Cache webových aplikací **3**
1.1 Úloha a problematika cache 3
1.1.1 Motivace použití webové cache 3
1.1.2 Poměr zásahů do mezipaměti 4
1.1.3 Zásady uvolňování 5
1.1.4 Expirace cache 5
1.2 Úrovně webové cache 5
1.2.1 Cachování na straně klienta 6
1.2.2 Síť doručování obsahu (CDN) 6
1.2.3 Cachování reverzní proxy 6
1.2.4 Cache webové aplikace 7
1.3 Návrhové vzory 7
1.3.1 Cache-Aside 7
1.3.2 Read-Through 8
1.3.3 Write-Through 9
1.3.4 Write-Behind 9
1.3.5 Write-Around 10
1.3.6 Refresh-Ahead 10
1.3.7 Shrnutí 11

2 Distribuovaná cache **12**
2.1 Srovnání s in-memory cache 12
2.1.1 Přednosti in-memory cache 12
2.1.2 Nedostatky in-memory cache 12
2.2 Charakteristika distribuované cache 13
2.2.1 Výhody distribuované cache 14
2.2.2 Problematika distribuované cache 15
2.3 Topologie distribuovaných cache systémů 17
2.3.1 Replikovaná topologie 17
2.3.2 Parciální topologie 18
2.3.3 Kombinace 19
2.3.4 Blízká mezipaměť 20

3 Cachování v prostředí ASP.NET Core **22**
3.1 Prostředí .NET Core a ASP.NET Core 22
3.2 Typy cache v prostření ASP.NET Core 22
3.2.1 Cachování typu in-memory 22
3.2.2 Distribuované cachování 23
3.2.3 Cachování v Razor pohledech 24
3.2.4 Cachování odpovědi 24
3.2.5 Cachování výstupu 24

4 Cache mechanismy v Kentico Xperience **26**
4.1 Kentico Xperience 26
4.1.1 Architektura 26
4.1.2 Webové farmy 27
4.2 Závislostní klíče 29
4.3 Typy cache 29
4.3.1 Cachování dat 29
4.3.2 Rozšíření cachování v Razor pohledech 31
4.3.3 Cachování souborů 32
4.4 Synchronizace cache v rámci webové farmy 33

5 Požadavky na cache modul **34**
5.1 Funkční požadavky 34
5.1.1 Integrace do Kentica Xperience 34
5.1.2 Závislosti záznamů 34
5.1.3 Priority záznamů pro uvolňování paměti 34
5.1.4 Metody po vymazání záznamu z cache 34
5.1.5 Zachování expirace záznamů 35
5.1.6 Synchronizace cache v rámci web farmy 35
5.1.7 Administrativní rozhraní pro správu cache . . . 35
5.1.8 Trvalost cache 35
5.1.9 Podpora závislostních klíčů při cachování v Razor pohledech 36
5.2 Nefunkční požadavky 37
5.2.1 Škálovatelnost 37
5.2.2 Zabezpečení 37
5.2.3 Snadná integrace do existujícího systému 37
5.2.4 Dodržení principu oddělení zodpovědnosti . . . 37

6 Analýza cache systémů **38**
6.1 Memcached 38
6.2 Redis 40
6.3 Hazelcast 43
6.4 Shrnutí analýzy 45
6.5 Výběr vhodného systému 48

7 Implementace a kódová dokumentace **49**
7.1 Použité technologie 49
7.2 Analýza a návrh 49
7.2.1 Návrhový vzor cache mechanismu 49
7.2.2 Analýza architektury nativního cachovacího mechanismu 50
7.2.3 Diagram tříd 51
7.3 Implementační detaily služby pro práci s Redis 52
7.3.1 Použité knihovny při implementaci 53
7.3.2 Funkce závislosti na fiktivním klíči 53
7.3.3 Priorita cache záznamu 55
7.3.4 Podpora akce po smazání 56
7.3.5 Podpora klouzavé expirace 58
7.4 Rozšíření tag helperu distribuované cache 58
7.5 Serializace objektů 58
7.6 Administrační rozhraní 59
7.7 Registrace služeb 60
7.8 Metodika testování a ověření funkčnosti 60
7.9 Integrace modulu do projektu 61

Závěr **63**

A Elektronická příloha **65**

Bibliografie **66**

Seznam tabulek

4.1 Ukázka fiktivních klíčů 29
6.1 Srovnávací tabulka 47
7.1 Návrh způsobu uchovávání závislostních klíčů 54
7.2 Nastavené prahové hodnoty 56

Seznam obrázků

1.1 Úrovně cache v rámci webové architektury 6
1.2 Vzor Cache-Aside 8
1.3 Vzor Read-Through 8
1.4 Vzor Write-Through 9
1.5 Vzor Write-Behind 10
1.6 Vzor Write-Around 10
1.7 Vzor Refresh-Ahead 11

2.1 Architektura integrace distribuovaného cache systému . . 14
2.2 Replikovaná topologie distribuovaného cache systému . . 18
2.3 Parciální topologie distribuovaného cache systému 19
2.4 Příklad kombinované topologie 20
2.5 Topologie Blízká mezipaměť 21

4.1 Kentico Xperience architektura [31] 27
4.2 Infrastruktura s webovou farmou 28

5.1 Diagram případů užití 36

6.1 Příklad Redis Sentinel [27] 41
6.2 Hazelcast replikace 44

7.1 Struktura současného cachovacího mechanismu 50
7.2 Diagram tříd modulu distribuované cache 52
7.3 Rozhraní správy distribuované cache v administraci Kentica 60

Úvod

V dnešní době, kdy nároky na rychlost a efektivitu webových aplikací neustále rostou, je klíčové hledat inovativní řešení, která optimalizují výkon a uživateli zajistí plynulou interakci. Důležitým prvek této optimalizace je efektivní využití webové cache. Cache představuje mechanismus, který dočasně uchovává často přistupovaná data za účelem snížení latence a zrychlení odezvy na požadavky uživatelů. Osvědčený a běžně využívaný přístup je ukládat cachovaná data přímo na serveru, na kterém je aplikace provozována. Nicméně v některých případech může tato metoda narazit na omezení, zejména pokud jde o zpracování rostoucího objemu dat a požadavků na rychlý přístup. V těchto situacích nabízí distribuovaná cache významné vylepšení, poskytující škálovatelné a efektivní řešení pro správu cache ve webových aplikacích.

Platforma Kentico Xperience se řadí mezi přední systémy správy obsahu postavených na technologii .NET. Platforma je specifická svou architekturou a cachovacími mechanismy. Nicméně v současné době neexistuje žádné dostupné řešení, které by poskytovalo efektivní integraci distribuované cache přímo do této platformy. Hlavním cílem této práce je navrhnout a implementovat vlastní modul, který dokáže integrovat technologii distribuované cache paměti do platformy Kentico Xperience.

Tato práce je rozdělena do sedmi částí. První kapitola představuje fundamentální koncepty webového cachování jako takového. Ve druhé kapitole je provedeno porovnání mezi lokálním a distribuovaným přístupem cachování. Třetí kapitola se zabývá konkrétními způsoby implementace těchto přístupů v prostředí ASP.NET. Čtvrtá kapitola analyzuje stávající mechanismy cachování v rámci platformy Kentico Xperience. Na základě předcházejícího rozboru, v páté kapitole stanovuji požadavky na vznikající modul. V šesté kapitole provádím analýzu a srovnání systémů Memcached, Redis a Hazelcast, coby poskytovatelů distribuované cache. Poslední kapitola provádí procesem návrhu a implementace samotného modulu. V kapitole jsou zahrnuty především implementační detaily zajímavých problémů, které bylo třeba řešit.

Výstupem praktické části práce je plně implementovaný a funkční modul integrující technologii distribuované cache do platformy Kentico Xperience, který se podle popsaných instrukcí může začlenit do existujícího systému.

Diplomová práce vznikla ve spolupráci se společností zaměřující se na vývoj webových aplikací postavených na platformě Kentico – BiQ Bluesoft a. s. Účelem práce bylo vypracování analýzy ohledně možností integrace technologie distribuované cache do platformy Kentico Xperience.

1 Cache webových aplikací

1.1 Úloha a problematika cache

Cache, neboli mezipaměť, je komponenta určena k ukládání dočasných dat. Jedná se o menší, rychlejší a dražší druh paměti, kam jsou nedávno či často používaná data ukládána za účelem celkového zlepšení výkonu při jejich načítání [1].

Existuje několik druhů cache. V této práci se budu zabývat cachí webových aplikací.

1.1.1 Motivace použití webové cache

Hlavní přínosy cache paměti lze shrnout následovně [2, 3, 4]:

Zlepšuje výkon aplikací
Cache webových aplikací je ukládána do paměti typu `RAM` (random-access memory). Při porovnaní časové nákladovosti operace čtení z databáze, je čtení dat z mezipaměti velmi rychlé. Tento výrazně rychlejší přístup k datům snižuje latenci a zvyšuje celkový výkon aplikace.

Snižuje náklady na databázi
Díky své rychlosti, mezipaměť může obsloužit velké množství vstupně-výstupních operací za sekundu (z anglického *input-output operations per second* – `IOPS`). Na druhou stranu, databázový systém dovede obsloužit několikanásobně menší počet takových operací. Při správném nastavení tak jediná instance mezipaměti může nahradit několik instancí databáze a snížit tak celkové náklady na provoz systému.

Snížení výpočetní zátěže
Uložením výsledků opakovaných a složitých výpočtů do cache paměti se snižuje celková zátěž na výpočetní jádro serveru. Zároveň přesměrováním významné části čtecí zátěže z databáze do paměti se sníží celková zátěž na databázi.

Zmírnění zatížení serveru v době špiček
Zvládání náhlých výkyvů ve využívání aplikace je častou výzvou v moderních aplikacích. Zvýšené zatížení serveru má za následek vyšší latenci při získávání dat, což způsobuje nepředvídatelnost celkového výkonu aplikace. Využitím cache lze tento problém zmírnit.

Eliminace databázových horkých míst
Horké místo v databází (z anglického *database hotspot*) je malá podmnožina dat, která je v porovnání s jinými částmi velmi často používaná. To může vyžadovat nadměrné poskytování databázových prostředků na základě požadavků na propustnost nejčastěji používaných dat. Uložení takových dat do mezipaměti zmírňuje potřebu nadměrného přidělování prostředků a zároveň zajišťuje rychlý a předvídatelný výkon pro nejčastěji používaná data.

1.1.2 Poměr zásahů do mezipaměti

Při požadavku o získání dat z cache mohou nastat dvě situace:

- **Zásah cache** (*cache-hit*) označuje úspěšné nalezení dat v cachi.
- **Minutí cache** (*cache-miss*) signalizuje fakt, že požadovaná data se v cachi nenachází. Reakce na tuto odpověď závisí na konkrétním návrhu cachovacího mechanismu. Povětšinou dojde k načtení dat provedením výpočetní operace nebo databázového dotazu a následnému uložení výsledku do cache.

Poměr zásahů do mezipaměti (z anglického *cache hit ratio*) je pak poměr mezi počtem zásahů a celkového počtu požadavků o získání dat.

$$\text{poměr zásahů} = \frac{\text{počet zásahů}}{\text{počet zásahů} + \text{počet minutí}}$$

Pro efektivní využití cache by měl být poměr zásahů do cache co nejvyšší. Při nízkém poměru zásahů dochází ke dvěma negativním dopadům:

- Většina požadavků mine cache a operace pro načtení dat se provede pokaždé.

- Nízký poměr zásahů do mezipaměti vede k velkému využití celkové kapacity mezipaměti, protože každé minutí znamená nový zápis do cache.

U takových případů je potřeba zvážit, zda je cachování opravdu potřeba [5, 6].

1.1.3 Zásady uvolňování

Velikost cache paměti není neomezená. Nastavení její kapacity je součástí konfigurace cachovacího mechanismu. Při zavedení cache tedy musíme brát v potaz i tzv. zásady uvolňování z cache – tedy nastavení postupů, podle kterých se zareaguje na případ zaplněného cache úložiště. Nejčastěji se setkáme s variantami, které vybírají záznamy k uvolnění podle algoritmů `LRU` a `LFU` [7].

- **LRU** (*Least recently used*) – z cache jsou uvolněny **nejdéle nepoužívané** záznamy.

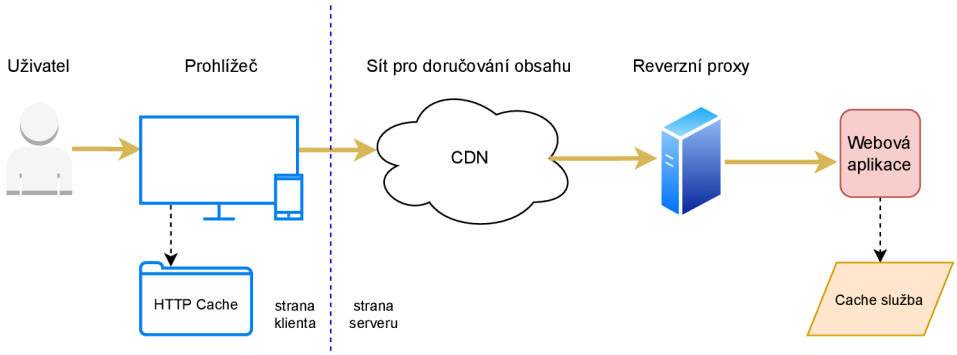
- **LFU** (*Least frequently used*) – z cache jsou uvolněny **nejméně používané** záznamy.

1.1.4 Expirace cache

Expirace stanovuje dobu platnosti záznamu v cachi. Po vypršení platnosti dojde ke smazání záznamu z cache. Expirace se může specifikovat **absolutním časem** nebo **klouzavou hodnotou**. Absolutní vypršení platnosti nastavuje expiraci na konkrétní čas a pomáhá předejít dlouhotrvajícím nekonzistencím mezi cachovanými a aktuálními daty. Expirace s klouzavou hodnotou funguje na principu opětovného nastavování expirace po každém přístupu k datům [7].

1.2 Úrovně webové cache

Cache se ve webových systémech může integrovat na různých úrovních napříč síťovou architekturou [8]. To je vidět na obrázku 1.1.



Obrázek 1.1: Úrovně cache v rámci webové architektury

1.2.1 Cachování na straně klienta

Nejzákladnější úrovní je ukládání do cache na straně klienta, kdy uživatel webu využívá vestavěnou `HTTP` cache integrovanou v prohlížeči. Jedná se o opatření, které lze použít ke snížení latence při vyžádání zdrojů z webového serveru. Metodika ukládání do mezipaměti je založena na nastavení `HTTP` hlavičky odpovědi serveru. Moderním přístupem je použití hlavičky `Cache-Control` pomoci které se dá specifikovat expirace a další nastavení [9].

1.2.2 Síť doručování obsahu (CDN)

Síť doručování obsahu (z anglického *content delivery network*) ukládá statický webový obsah (například obrázky, videa nebo webové stránky) do cache v rámci sítě spravovaných proxy serverů. Příchozí požadavek je zpracován uzlem blízký klientovi, kterému je zaslán cachovaný obsah – díky tomu jsou CDN servery schopny doručit obsah rychleji než webový server [10].

1.2.3 Cachování reverzní proxy

Reverzní proxy server je podobný klasickému (forward) proxy serveru – oba fungují jako prostředník mezi klientem prohlížeče a webovým serverem. Hlavní rozdíl je v tom, že klasický proxy server je umístěn blíže ke klientovi a reverzní proxy server je umístěn blíže

k webovému serveru. Jelikož jím procházejí požadavky a odpovědi, může reverzní proxy server uložit odpověď serveru do cache a v budoucnu ji použít k odpovědi na další požadavky, aniž by při tom komunikoval se serverem webové aplikace [11].

1.2.4 Cache webové aplikace

V prostředí webové aplikace se do cache ukládá pomocí dvojice klíč-hodnota. Na rozdíl od cachování u proxy serverů, které ukládají celé `HTTP` odpovědi, tento přístup lze použít k ukládání jakéhokoli webového obsahu do mezipaměti podle potřeby vývojáře aplikace. Do mezipaměti se dají umístit odpovědi databázových dotazů, výsledky výpočtů, obsahy celých stránek nebo informace k webovým relacím. Cachovaná data mohou být uložena přímo v hlavní paměti webového serveru nebo prostřednictvím externí cachovací služby.

1.3 Návrhové vzory

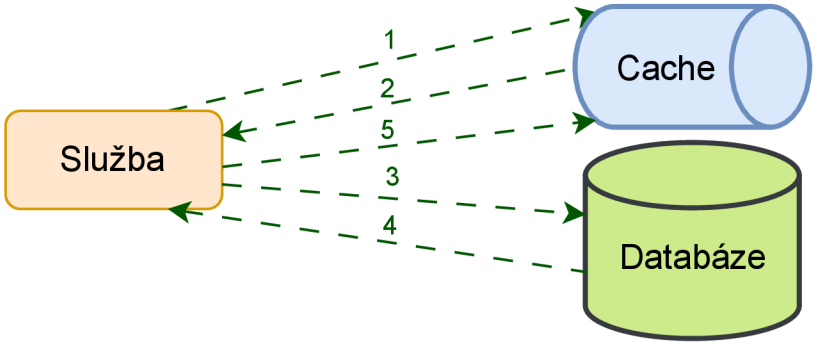
Při integraci cachovacího mechanismu do aplikace je důležitým krokem navrhnout způsob jakým bude aplikace a cache vzájemně spolupracovat. Tato část práce se zaměřuje na možné vzory, jakými je možné tuto interakci provozovat [12, 7].

1.3.1 Cache-Aside

Jedná se o nejběžnější strategii ukládání do cache. Odpovědnost za řízení toku mezi cachí a datovým zdrojem přebírá aplikační kód. Používá se reaktivní přístup, kdy se data načítají do cache líně (z anglického *lazy-loading*) – do cache se data nahrají až ve chvíli první potřeby. Největší výhodou této strategie je její jednoduchost a přehlednost, kdy požadavky na poskytovatele cache jsou poměrně nízké. To případně dovoluje velmi přímočarou migraci mezi různými poskytovateli. Jako nevýhodu lze považovat nutnost zvládnutí nekonzistence dat mezi cachí a databází přímo v kódu [13].

Dle diagramu 1.2 – Při potřebě načtení určitých dat se aplikace nejdřív dotáže cache (operace 1) a ta odpoví výsledkem dotazu (operace 2). Pokud se v cachi hledaný objekt nenachází (cache-miss),

aplikace načte data z databáze (operace 3 a 4). Následně se aktualizuje cache pro budoucí načtení (operace 5).

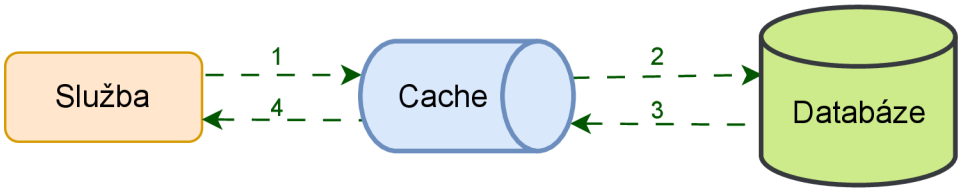


Obrázek 1.2: Vzor Cache-Aside

1.3.2 Read-Through

Tento přístup přenáší odpovědnost za získání hodnoty z datového úložiště na poskytovatele cache. Aplikační kód nyní komunikuje pouze s cachovací službou, která nyní spravuje synchronizaci mezi sebou a datovým úložištěm [7].

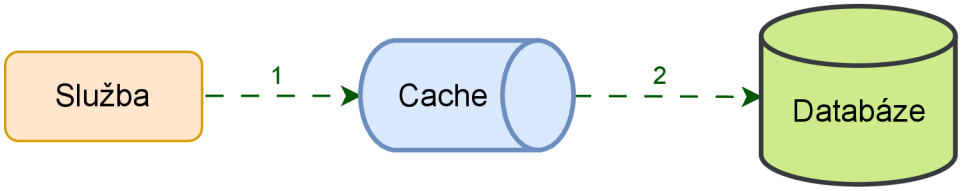
Podle diagramu 1.3 v případě nenalezení dat v cachi (operace 1), poskytovatel cache načte data z datového úložiště a umístí je do mezipaměti pro budoucí načtení (operace 2 a 3) a vrátí hodnotu zpět volajícímu (operace 4).



Obrázek 1.3: Vzor Read-Through

1.3.3 Write-Through

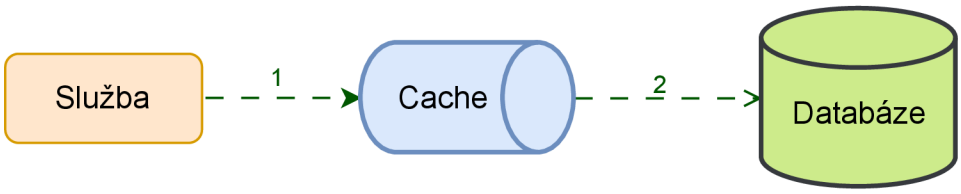
Podobný přístup jako v případě *Read-Through*, ale určený pro zápis. Při této technice zapisujeme data do datového úložiště prostřednictvím cache služby. To znamená, že data budou nejprve vložena/aktualizována v rámci cache a až poté v rámci datového úložiště (stejně jako na diagramu 1.4 v operacích 1 a 2). To pomáhá udržovat data v cachi konzistentní. Nicméně nevýhodou tohoto návrhu je pomalá operace zápisu, kdy je potřeba ukládat hodnoty do dvou úložišť (cache a databáze) [7].



Obrázek 1.4: Vzor Write-Through

1.3.4 Write-Behind

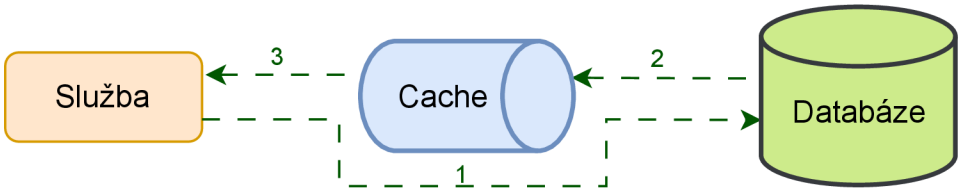
V tomto případě při pohledu na diagram 1.5 může být těžké všimnout si rozdílu s *Write-Through* přístupem. Stejně jako v minulém případě i zde zodpovědnost za zápis přebírá poskytovatel cache. Samotný zápis do datového úložiště ale probíhá asynchronně (znázorněno nevyplněnou šipkou operace 2). Do této chvíle u jiných návrhů byly všechny zprávy vyměňované mezi službou, cachí a databází synchronní: volající čeká až volaný dokončí zpracování a vrátí výsledek – a teprve poté pokračuje ve svém běhu. Při použití vzoru *Write-Behind* poskytovatel cache nečeká na potvrzení zapsání hodnoty do databázového úložiště. Tento přístup plní účel optimalizace výkonu aplikace snížením latence spojené s operacemi zápisu do datového úložiště. Na druhou stranu, případné chyby při zápisu nejsou odhaleny čímž může dojít k nekonzistenci [14].



Obrázek 1.5: Vzor Write-Behind

1.3.5 Write-Around

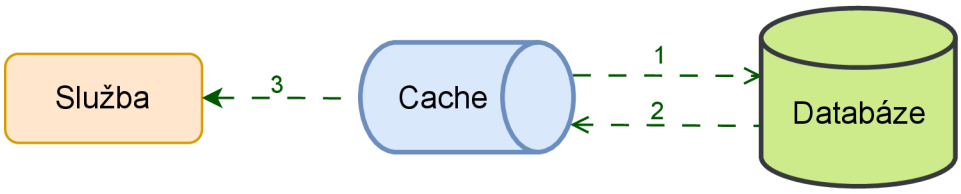
V tomto vzoru se při operaci zápisu cache obchází a zapisuje se přímo do datového úložiště (diagram 1.6 operace 1), zatímco operace čtení jsou nadále obsluhovány poskytovatelem cache (stejně jako u vzoru *Read-Through*). Jedná se o strategii, která upřednostňuje výkon při operaci čtení tím, že umožňuje operacím zápisu obejít poskytovatele cache. Díky tomu může být užitečná ve scénářích, kdy je omezená kapacita cache úložiště lépe využita pro často přístupná čtená data a kdy operace zápisu není primárním cílem optimalizace [7].



Obrázek 1.6: Vzor Write-Around

1.3.6 Refresh-Ahead

Čtení z datového úložiště je nákladná záležitost. Tento vzor se snaží předejít zvýšené latenci při operaci čtení tím, že se data v cachi asynchronně obnovují ještě před vypršením jejich platnosti – to je vidět na diagramu 1.7 (operace 1 a 2). Data se v cachi aktualizují před jejich použitím a to pomáhá snížit latenci [14, 7].



Obrázek 1.7: Vzor Refresh-Ahead

1.3.7 Shrnutí

Softwarový architekt vybere vhodný vzor v závislosti na konkrétních možnostech, dostupných zdrojích a požadavcích. Mezi dvě nejběžnější varianty se řadí přístupy *Cache-Aside* a *Write-Through*.

2 Distribuovaná cache

V minulé kapitole jsem uvedl motivace a obecnou problematiku cache ve webových aplikacích. V této kapitole se budu věnovat jejímu konkrétnímu technickému řešení – distribuované cachi. Jako motivaci pro zamyšlení se o potřebě distribuované cache nejdříve provedu srovnání s lokální cachí typu in-memory. Nejdříve popíšu její hlavní aspekty a uvedu hlavní výhody stejně tak jako některé její nedostatky. V druhé části kapitoly se zaměřím na základní charakteristiku distribuované cache a zvážím možné výzvy spojené s tímto přístupem.

2.1 Srovnání s in-memory cache

2.1.1 Přednosti in-memory cache

Jak z názvu vyplývá u cache typu in-memory jsou objekty ukládány přímo v paměti webového serveru. Implementace tohoto typu cache je relativně rychlá a nenáročná. Data se do cache ukládají v podobě reference – díky tomu je možné uložit objekt jakékoli struktury.

V případě, že se u existujícího projektu zjistí potřeba cachování, je často, díky své jednoduchosti, jako první implementována právě in-memory cache. Na rozdíl od externích cache úložišť, nejsou spojeny s žádnou další provozní režií, takže jejich integrace do systému je poměrně málo riziková.

Jelikož jsou data uložena přímo v operační paměti serveru je při načítání z cache předpokládána velmi nízká latence [15].

2.1.2 Nedostatky in-memory cache

I přes tyto významné výhody, in-memory cache se potýká s mnoha nedostatky.

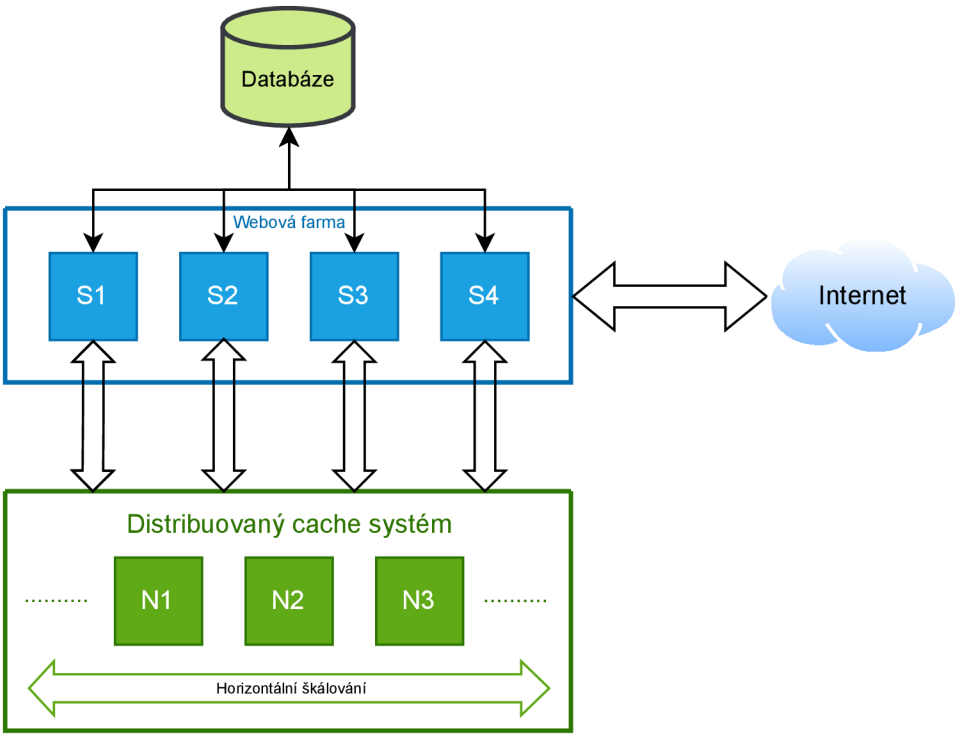
Běh cachovacího úložiště je přímo spojen s během webového serveru a tedy vypnutím aplikace cachovaná data zanikají. Tím dochází k tzv. studenému startu – tedy ke stavu, kdy úložiště cache je prázdné. Během doby jeho opětovného naplnění dochází k pomalejší odezvě systému a může dojít k přetěžování výpočetního výkonu či databáze [16].

Jiný problém nastane v případě provozu serverové farmy (aplikace běžící na více serverech) kdy může dojít k nekonzistencím mezi za sebou jdoucími požadavky. Pokud klient volá službu opakovaně, může nastat situace, kdy při prvním zavolání získá data novější a při prvním dalším volání obdrží data starší a již neaktuální. To se děje v závislosti na tom, který uzel ze serverové farmy požadavek zrovna zpracoval. Každý server v rámci serverové farmy totiž pracuje s vlastní verzí cache úložiště a tyto verze si většinou servery mezi sebou nijak nesynchronizují. Řešením takového problému by bylo zajistit, aby uživatelské relace byly pevné (z anglického *sticky session*). Takové relace zajišťují, že v jejich rámci jsou všechny požadavky odeslány na stejný server a nemůže tak dojít k těmto nekonzistencím v rámci jednotlivých volání [3].

S lokálním cachováním je spojena i nemožnost horizontálního škálování cachovací služby. Navýšením počtu serverů sice dosáhneme větší robustnosti systému jako celku, nicméně na kapacitu cachovacího úložiště to vliv mít nebude.

2.2 Charakteristika distribuované cache

Distribuovaná cache je typ cache, která je sdílena mezi několika servery. Je obvykle udržována jako externí služba pro aplikace, které k ní přistupují z jiných serverů. To je vidět na obrázku 2.1.



Obrázek 2.1: Architektura integrace distribuovaného cache systému

2.2.1 Výhody distribuované cache

Mezi hlavní výhody v porovnání s in-memory cache patří [17, 18]:

Vylepšená dostupnost
Rozložení cache paměti do více uzlů pomáhá zlepšit její dostupnost. Řeší problém jediného bodu selhání (z anglického *Single point of failure*) – Pokud dojde k výpadku v jednom ze serveru, většina paměti zůstane stále dostupná [19].

Zároveň zde odpadá problém studeného startu během restartu webového serveru. Běh aplikace nyní nemá na běh cachovacího systému žádný vliv.

Škálovatelnost
Možnost horizontální škálovatelnost je přímo v povaze distribuova-

ných systémů. V případě potřeby přizpůsobení výkonu distribuovaného cache systému je možné serverové uzly jednoduše přidat nebo odebrat.

Robustnost cache
Pro jednotlivé uzly cache systému je možné nastavit pravidelné zálohy na disk. Díky tomu se uzel dokáže po výpadku rychleji vzpamatovat.

Některé cachovací systémy podporují pokročilé mechanismy schopné detekovat výpadky a automaticky jim předcházet (toto téma detailně rozebírám v sekci 2.3).

Nastavitelná konzistence
Míra konzistence v distribuovaném cache systému je nastavitelná. Možné úrovně konzistence se mezi konkrétními cache systémy liší. Nejčastěji se setkáme s tzv. **eventuální** nebo **silnou** konzistencí. Eventuální konzistence garantuje, že provedené změny se během krátkého časového období samy propíšou do všech ostatních uzlů systému, a nakonec tak budou navzájem konzistentní. Silná konzistence zaručuje, že každá operace čtení vrátí výsledek poslední operace zápisu.

Uvolnění lokální paměti
Přesunutím úložiště cache z webového serveru do odděleného distribuovaného systému se uvolní výpočetní výkon pro samotné fungování aplikace.

2.2.2 Problematika distribuované cache

Ačkoli distribuovaná cache řeší některé nedostatky lokální in-memory cache, přináší s sebou nové nedostatky a výzvy, které je nutné v době návrhu vzít v potaz.

Serializace dat

Při přenosu dat do externího systému je nutné hodnoty převést do formátu, který bude po síti přenositelný – pole bytů. Tomuto typu konverze se říká serializace. Opačnou akcí, tedy konverze pole bytů na původní objekt se nazývá deserializace. Návrh efektivního způsobu serializace je klíčový během integrace distribuovaného cache systému.

Během operace cachování, serializace a deserializace představují dodatečnou nutnou režii, která se může negativně projevit na výkonu celé aplikace [20, 21].

Síťová komunikace

Síťový přenos je další faktor, který způsobuje značné omezení během práce s distribuovaným systémem. Při návrhu je dobré ověřit limity šířky pásma, propustnost paketů, spolehlivost a zpoždění sítě. Při přenosu citlivých dat je potřeba počítat i s režijními náklady šifrovacích funkcí jako `TLS/SSL` [7].

Zabezpečení

Pokud webová aplikace pracuje s citlivými daty, je nutné vzít v potaz zabezpečení integrace systému externí cache. Základním požadavkem je zamezit neautorizovaný přístup k datům na externím úložišti. Dosáhnout toho lze skrytím instance služby pro veřejnost, nastavením firewallu a konfigurací systému řízení přístupu (např. `ACL` [1]) pro konkrétní službu. Jako ochrana před odposloucháváním komunikace může být síťová komunikace zašifrovaná pomocí `TLS/SSL`. Pro zachycení nežádoucích činností je možné nastavit monitorování aktivit [23, 24, 15, 7].

Případ výpadku

Během návrhu je také dobré vytvořit plán akcí v případě chyby při komunikaci s externí cache službou. Při výpadku externí služby existuje celá řada možných strategií. V kontextu distribuovaného cache systému se nabízejí tyto tři strategie [25]:

- Opakování akce – ihned nebo po určité odmlce
- Proaktivní opakování akce – znovu provést akci, ale tentokrát paralelně několikrát a použít výsledek první dokončené

1. Access Control List – sada dat informující operační systém počítače, který následně uděluje uživatelům nebo skupině uživatelů přístupová práva k určitému objektu v systému, například k souboru nebo adresáři [22].

- Failover [2] – proces, při kterém se, v případě selhání hlavního systému nebo komponenty, automaticky přepíná na záložní systém nebo komponentu

Kromě plánování reakce na výpadky, je dobré zkoumat příčiny těchto výpadků a těm se snažit předejít.

2.3 Topologie distribuovaných cache systémů

Distribuované cache systémy mohou být tvořeny jedním nebo více servery. Uskupení více serverů, které dohromady tvoří jednu logickou jednotku se nazývá *počítačový cluster*. Uzly v clusteru spolu navzájem komunikují a synchronizují se prostřednictvím sítě [26].

Topologie cache systému definuje jeho interní chování, míru spolehlivosti a rychlost. Jednotlivé cache systémy mohou podporovat různé typy topologií a při výběru vhodného cache systému je dobré tuto skutečnost brát v potaz. V rámci této části práce popíšu několik základních topologií [7, 26, 27, 28, 29].

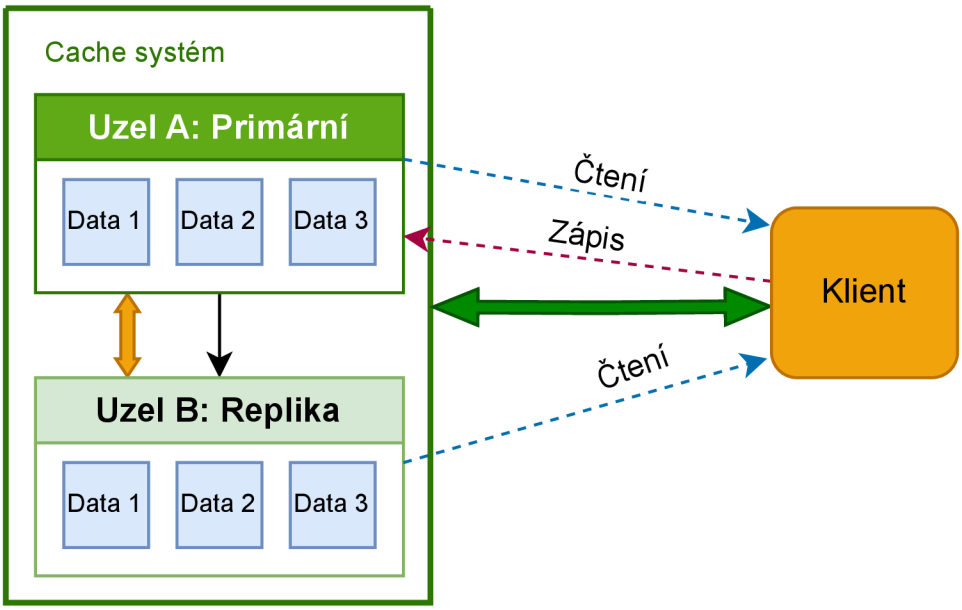
2.3.1 Replikovaná topologie

V replikované topologii všechny uzly v clusteru obsahují celé cache úložiště. Jeden z uzlů je zvolen jako primární. Přes ten probíhají všechny zápisy do cache. Zápisy se pak propagují synchronně nebo asynchronně (závisí na konkrétních systému a konfiguraci) do zbývajících uzlů (tzv. replik). V případě, že se určitá část uzlů shodne, že primární uzel postihl výpadek, je provedeno přepnutí jedné z replik na primární uzel. Tím je zajištěna dodatečná dostupnost a spolehlivost. Tato topologie je vhodná v případech, kdy klientská aplikace potřebuje provádět mnoho operací čtení. Čtecí zátěž totiž může být rozprostřena napříč všemi uzly clusteru – tedy i replikami. Garantovaná míra konzistence v takovém případě závisí na nastavené úrovni konzistence (viz 2.2.1). Replikovaná cache není ideální topologií pro operace náročné na zápis.

Na diagramu 2.2 je vidět příklad takové topologie s dvěma uzly. Uložená data jsou obsažena v obou uzlech. Ty spolu udržují spojení

2. Slovo pochází z anglického jazyka a do češtiny se nepřekládá

za účelem ověření vzájemné dostupnosti. V případě zápisu, primární uzel propaguje změnu do repliky. Klient stále komunikuje s cache systémem prostřednictvím rozhraní jako s jedním logickým celkem. Operace čtení může být fyzicky obsloužena i replikovaným uzlem.



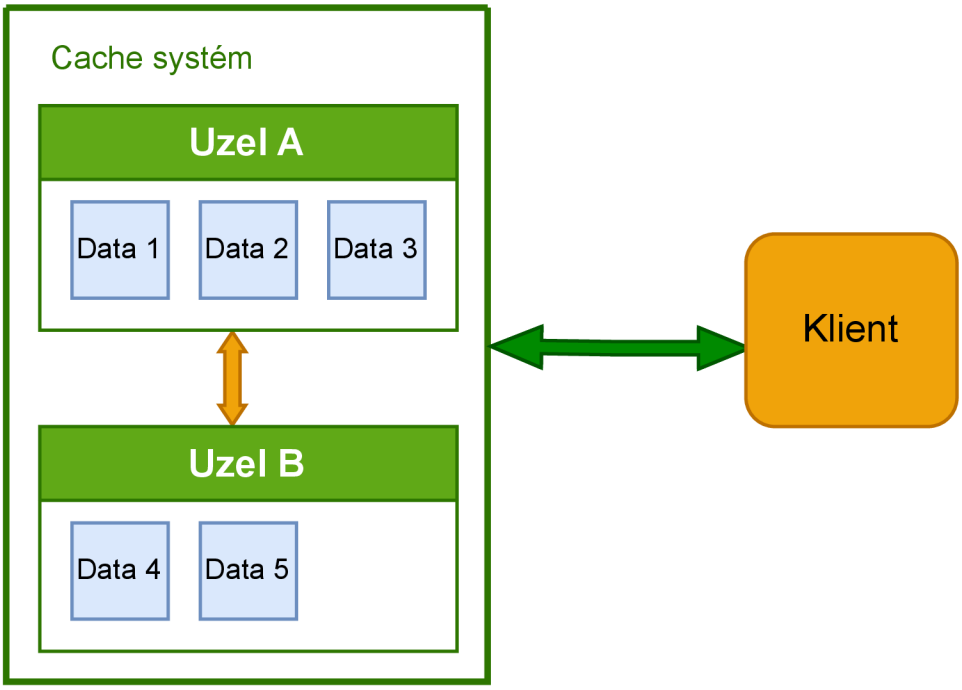
Obrázek 2.2: Replikovaná topologie distribuovaného cache systému

2.3.2 Parciální topologie

Parciální topologie rozdělí úložiště cache na oddíly. Tomuto přístupu se říká *sharding*. Uzlům, které drží jednotlivé části úložiště se zase říká *shards*. Tato topologie je vhodná v případech, kdy operace čtení a zápisu jsou přibližně stejně časté. Přidáním dalších uzlů do clusteru se zvýší kapacita transakcí a kapacita úložiště cache systému. Celková kapacita je rovna součtu kapacit jednotlivých shardů.

S tímto přístupem musí být nastavena strategie mapování klíčů k jednotlivým shardům. Strategie se opět liší v závislosti na konkrétním systému distribuované cache.

Na diagramu 2.3 je vidět, jak jsou data rozdělena mezi dva shardy. Toto konkrétní rozvržení záznamy v cachi nijak neduplikuje, a tedy v případě výpadku jednoho ze serveru dojde ke ztrátě části dat.



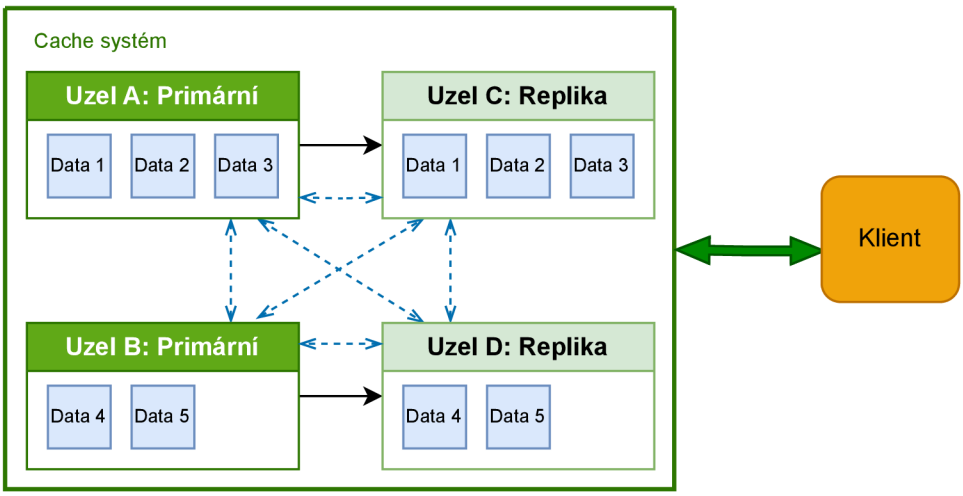
Obrázek 2.3: Parciální topologie distribuovaného cache systému

2.3.3 Kombinace

U moderních systémů se často využívá kombinace zmíněných topologií čímž se spojí výhody replikované a parciální topologie. Úložiště je tedy rozprostřeno napříč několika primárními uzly a ty jsou pak navíc replikovány. Oproti čistě parciální topologii je zde zajištěna ochrana proti výpadku a ztrátě dat. Na druhou stranu oproti čistě replikované topologii, je zde podporováno horizontální škálování kapacity úložiště.

Nevýhodou je komplexnější správa a složitější možnost rozšiřování stávajícího systému – spolu s novým primárním uzlem musíme zavést i jednu nebo více replik.

Na diagramu 2.4 je znázorněn cluster o čtyřech uzlech. Modré šipky znázorňují, jak všechny uzly navzájem udržují spojení mezi sebou. Při společné detekci výpadku primárního uzlu dojde k přepnutí jeho repliky na primární uzel.



Obrázek 2.4: Příklad kombinované topologie

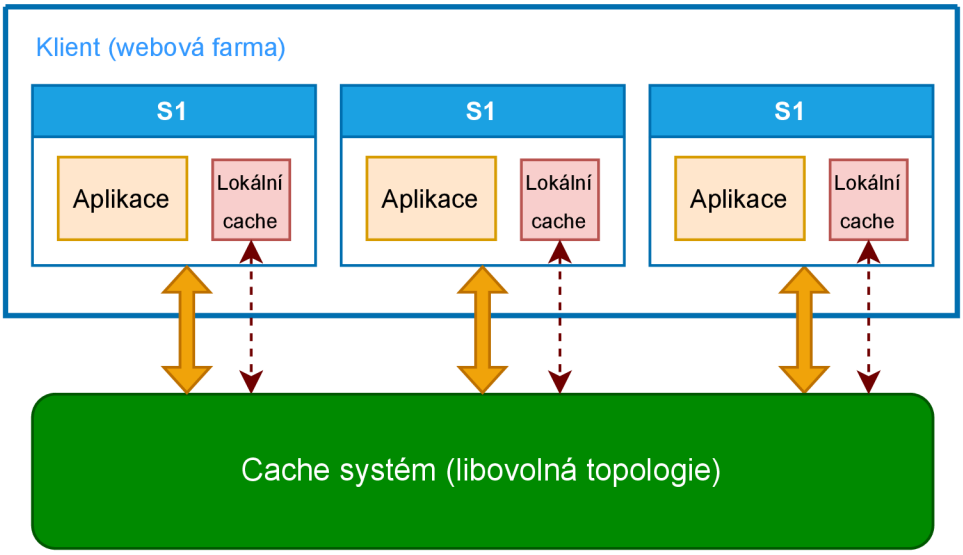
2.3.4 Blízká mezipaměť

Blízká mezipaměť (z anglického *near cache*), také známá jako *cache klienta*, je speciální typ topologie, která propojuje cache distribuovaného systému s lokální pamětí webového serveru. Jedná se tedy o hybridní druh cache.

Účelem této topologie je snížit zpoždění a zlepšit celkový výkon tím, že do menší lokální paměti se ukládají záznamy, ke kterým aplikace často přistupuje. Při opakujícím se dotazu tedy není potřeba žádného síťového přenosu mezi aplikačním serverem a distribuovanou cache službou. Výzvou u této topologie je dosažení určité úrovně konzistence mezi lokální a externí mezipamětí. Některé cache systémy přímo podporují synchronizaci s lokální cachí, jiné toto nechávají na vývojářích [30].

Na diagramu 2.5 jsou vidět komunikační kanály v rámci této topologie. Oranžové šipky znázorňují běžnou komunikaci s distribuova-

ným cache systémem. Červené šipky reprezentují synchronizaci mezi jednotlivými úrovněmi cache.



Obrázek 2.5: Topologie Blízká mezipaměť

3 Cachování v prostředí ASP.NET Core

ASP.NET Core je open-source framework pro vývoj webových aplikací a služeb vyvinutý společností Microsoft. Jelikož je platforma Kentico Xperience s tímto frameworkem úzce spjata [31, 32], v této kapitole zmíněné prostředí a jeho vestavěné cachovací mechanismy stručně popíšu.

3.1 Prostředí .NET Core a ASP.NET Core

Microsoft .NET Core je multiplatformní framework podporující vývoj různých typů aplikací. Ačkoli je primárně spojen s jazykem C#, je navržen tak, aby podporoval více programovacích jazyků [33].

ASP.NET Core je součástí .NET Core frameworku, která je zaměřena na vývoj webových aplikací a API. Podporuje technologie jako MVC [1], Razor Pages [2], Blazor [3], Web API a další [37].

3.2 Typy cache v prostření ASP.NET Core

Prostředí ASP.NET Core poskytuje několik přístupů, jakými je možné cachovací mechanismy v rámci aplikačního kódu aplikovat [38, 39].

3.2.1 Cachování typu in-memory

Základní funkce cachování přímo do paměti serveru je postavena okolo rozhraní `IMemoryCache`. Aktuální implementace pochází z balíčku `Microsoft.Extensions.Caching.Memory`, která je obalem kolem slovníku `ConcurrentDictionary<object, CacheEntry>`.

Záznamy v mezipaměti, reprezentované pomocí třídy `ICacheEntry`, mohou být objektem libovolného typu [39].

---

1. zkratka pro Model-View-Component – softwarová architektura, která odděluje datovou vrstvu aplikace, pohledy aplikace a řídicí logiku do tří nezávislých komponent [34]
2. Zjednodušený návrhový model. Převážná část logiky přesunuta do stránek pohledů [35]
3. Framework pro vytváření interaktivních webových aplikací [36]

`IMemoryCache` drží pro každý objekt tzv. slabý odkaz – objekt typu `WeakReference`. Jedná se o speciální typ odkazu, který je sběračem odpadu (z anglického *garbage collector*) při počítání referencí ignorován. V případě, že na objekt neexistují žádné silné reference a je pouze slabě odkazován, automaticky je uvolněný z hlavní paměti [40].

Prostředí ASP.NET Core poskytuje dodatečnou podporu pro několik funkcionalit ve spojení s in-memory cachováním. Prostřednictvím třídy `MemoryCacheEntryOptions` je možné definovat absolutní a klouzavou expiraci nebo prioritu, která je určující při nuceném uvolňování záznamů z cache v důsledku zaplněného úložiště. Dále je možné při vkládání záznamu do cache nastavit funkci, která se při vymazání tohoto záznamu z cache má vykonat (dále *akce po smazání*). Pomocí tzv. tokenu zrušení (z anglického *cancellation token*) se pak nastavují závislosti mezi cachovanými objekty [3].

3.2.2 Distribuované cachování

Pro použití distribuovaného cachování ASP.NET Core poskytuje rozhraní `IDistributedCache`. V porovnání s API pro lokální cachovaní, jsou funkce podporované tímto rozhraním značně omezené. Zatímco v případě cachování do paměti mohou být klíčem objekty libovolného typu, klíče u distribuovaného cachování musí být textový řetězec. Dále při ukládání do paměti může být hodnota libovolného silně typovaného generického typu. V distribuovaném prostředí jsou hodnoty uchovávány jako pole bytů. Třída `DistributedCacheEntryOptions` sice nabízí nastavení absolutní i klouzavé expirace, nicméně zde chybí nastavení priority, tokenů zrušení, expiračních tokenů a akce po smazání [39].

Microsoft vytvořil několik vlastních implementací pro rozhraní `IDistributedCache` [17]:

- Distributed Redis cache – Jako distribuovaný cache systém používá open source službu Redis.
- Distributed SQL Server cache – Využívá se jako zálohovací úložiště. Objekty se ukládají do samostatné SQL tabulky.
- Distributed Memory cache – Objekty ukládá do paměti webového serveru, a tedy ve skutečnosti se nejedná o distribuovanou cache. Využívá se hlavně při vývoji a testování.

3.2.3 Cachování v Razor pohledech

CacheTagHelper

Cachování přímo v Razor pohledech je umožněno pomocí tag helperu[4] `cache`, který pro ukládání dat používá in-memory cachování.

```
<cache vary-by-query="param">
    @DateTime.Now
</cache>
```

Tag helper podporuje atributy pro řízení expirace a uchovávání různých verzí výstupů v závislosti na hlavičce výstupu (`vary-by-header`), parametru dotazu (`vary-by-query`), cestě `URL` (`vary-by-route`) a dalších případů [41].

DistributedCacheTagHelper

Jako podporu pro distribuované cachování Microsoft představil tag helper `distributed-cache`, který dědí všechny původní atributy. Pro ukládání dat využívá rozhraní `IDistributedCache` [42].

```
<distributed-cache name="unique_cache_key" vary-
   by-query="param">
    @DateTime.Now
</distributed-cache>
```

3.2.4 Cachování odpovědi

Pomocí atributu `[ResponseCache]` ASP.NET Core umožňuje nastavit cachování na straně klienta. Docílí toho úpravou polí (`Vary`, `Cache-Control` a jiné) hlavičky odpovědi. Na základě toho si webový prohlížeč uchová odpověď webového serveru pro další použití [43].

3.2.5 Cachování výstupu

Funkce cachování výstupu (z anglického *output caching*), představená v ASP.NET Core ve verzi 7, umožňuje cachovat odpovědi na straně webového serveru.

4. Jedná se o technický termín specifický pro ASP.NET Core a Razor Pages a do češtiny se nepřekládá

Pro ukládání odpovědí se používá rozhraní `IOutputCacheStore`. Výchozí implementace pracuje s lokální pamětí serveru, nicméně je možné vytvořit vlastní implementaci [44]. Ve verzi ASP.NET Core 8.0 Microsoft zveřejnil implementaci využívající jako úložiště službu Redis [45, 46].

4 Cache mechanismy v Kentico Xperience

Jelikož hlavní výstupem této práce má být implementace vlastního modulu určeného pro platformu Kentico Xperience, v této kapitole systém v krátkosti představím a popíšu cachovací mechanismy obsažené v jeho základní verzi.

4.1 Kentico Xperience

Kentico CMS se řadí mezi komerční systémy správy obsahu (z anglického *content management system*). Platforma byla vytvořena českou společností Kentico Software. Kromě funkcí pro správu obsahu, systém poskytuje nástroje pro nastavení personalizace, online marketing, e-komerce nebo social networking. *Kentico Xperience* je název pro 13. verzi tohoto systému. Poslední 14. verze nazvaná *Xperience by Kentico* je oproti předchozím verzím postavená na zcela nové architektuře. Na vývoji této verze se stále pracuje a v tuto chvíli stále postrádá mnoho z důležitých funkcí z minulých verzí [47, 48].

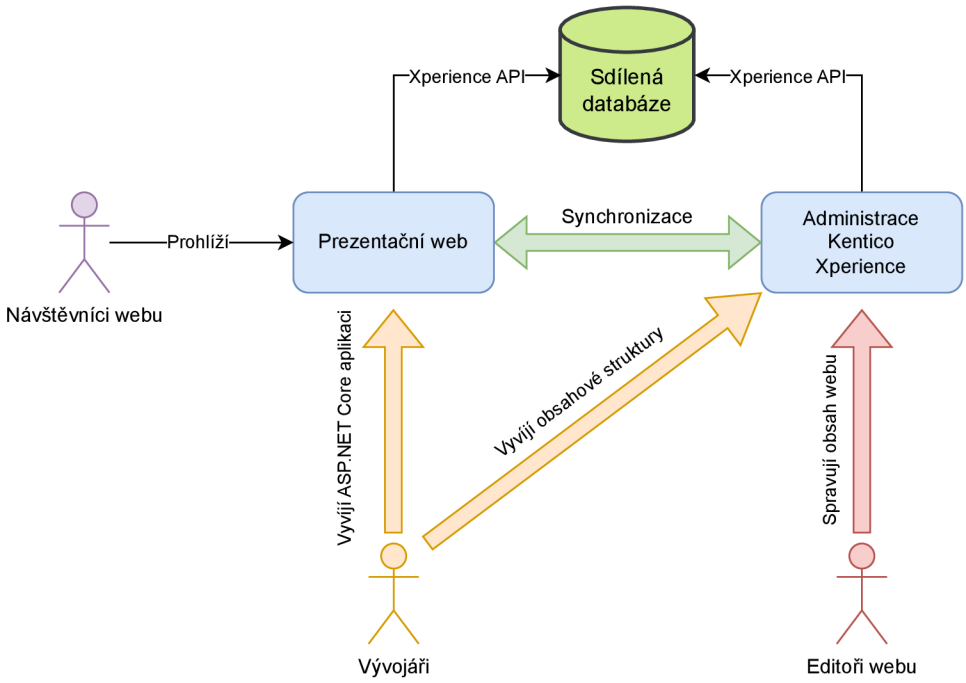
Kentico Xperience je založeno na technologiích .NET a Microsoft SQL Server. Poskytuje rozsáhlé API, které umožňuje vývojářům psát vlastní kód a rozšiřovat funkce systému. Lze vytvářet jak vlastní moduly a funkce, tak upravovat stávající mechanismy pro specifické potřeby [49].

4.1.1 Architektura

Typická architektura založená na platformě Kentico Xperience je složena ze dvou samostatných aplikací přistupujících k jednotné databázi. ASP.NET Core aplikace je zodpovědná za prezentační web (používá se označení livesite). Druhá aplikace, nazývána jako CMS, je postavena na technologii .NET Framework a slouží jako administrace a jako platforma pro plnění obsahu [31].

Jak je znázorněno na diagramu 4.1, k platformě přistupují tři hlavní aktéři. Návštěvníci webu přistupují rovnou k livesite aplikaci. Za její vývoj a údržbu zodpovídají vývojáři. Ti ve fázi implementace livesity mohou využít libovolné technologie podporované ASP.NET Core jako MVC, Razor Pages nebo Blazor. Vývojáři také vytvářejí obsahovou

strukturu administrace jako jsou stránkové typy nebo modulové třídy a rozhraní. Obsah webu vyplňují editoři prostřednictvím administrativního rozhraní. Synchronizace znázorněná mezi livesite aplikací a administrací je zajištěna funkcionalitou webové farmy (podrobněji popsáno v sekci 4.1.2).



Obrázek 4.1: Kentico Xperience architektura [31]

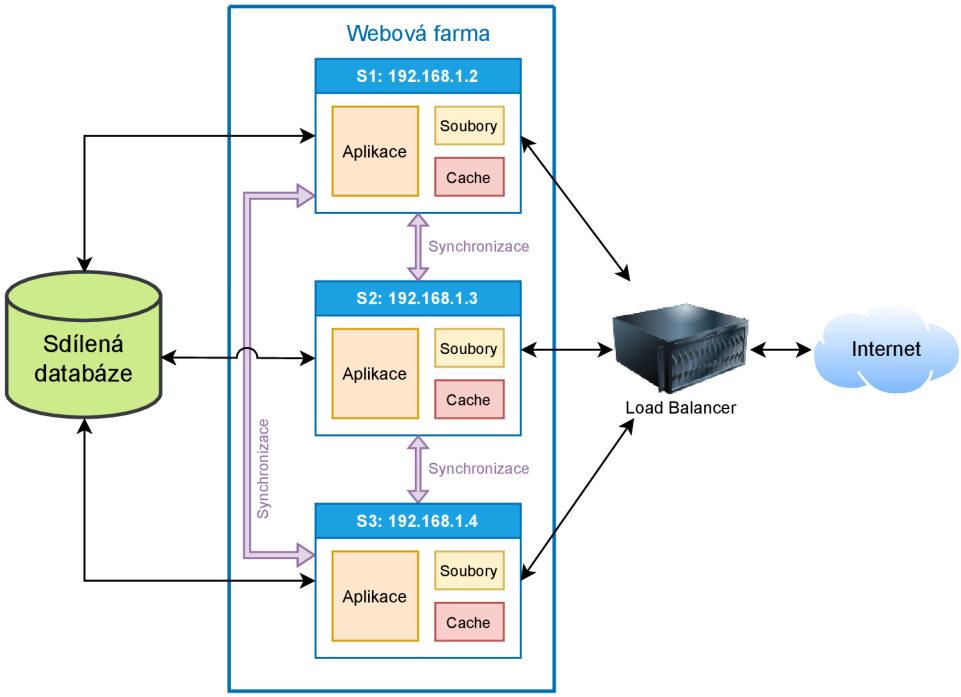
Na jedné instanci Kentico Xperience lze provozovat hned několik aplikací (tzv. site). Všechny sity se v rámci jedné instance připojují ke společné databázi a sdílí jeden webový projekt pro administrační rozhraní. Prezentační web sity může být tvořen samostatným projektem a běží na vlastní doméně [50].

4.1.2 Webové farmy

Kentico Xperience podporuje horizontální škálování formou webových farem. Každý přidaný server zvyšuje počet volání, které je systém schopný obsloužit. Zároveň výpadek jednoho ze serverů neznamená

výpadek celého systému. Mimo škálování, Xperience využívá tuto funkcionalitu pro synchronizaci mezi prezentačním webem a administrací.

Na obrázku 4.2 je vidět zjednodušený příklad infrastruktury s webovou farmou. Ve farmě se nacházejí celkem tři servery, každý s vlastní IP adresou, kopií souborů uložených v souborovém systému a cache úložištěm. Všechny tyto uzly přistupují k sdílené databázi. Volání klientů směruje jednotka vyrovnávače zátěže (z anglického *load balancer*). Vyrovnávač zátěže je samostatná komponenta, která se nachází před webovými servery a administrátoři systému jej konfigurují odděleně od platformy Kentico Xperience.



Obrázek 4.2: Infrastruktura s webovou farmou

Synchronizace mezi uzly probíhají formou zpráv nazývaných *úkol webové farmy* (z anglického *web farm task*). Tyto zprávy se ukládají do samostatné databázové tabulky `CMS_WebFarmServerTask` [51]. Každý server pak v samostatném vlákně spouští pravidelnou úlohu, která

vybere synchronizační úlohy určené pro daný server a ty pak spustí. Xperience se v základu stará o tyto synchronizace [52, 53]:

- synchronizace vymazaného cachovaného obsahu napříč webovou farmou
- synchronizace změn v nastavení sity
- synchronizace nahraných souborů (pouze v případě, že je pro uložení souborů využit souborový systém)

4.2 Závislostní klíče

Kentico používá tzv. fiktivní cachovací klíče (z anglického *dummy cache keys*) za účelem vytvoření závislostí mezí cachovanými daty a jinými objekty. Tyto fiktivní klíče neobsahují žádné data, ale pouze reprezentují jeden nebo více objektů. Pokud dojde ke změně některého z těchto objektů, dojde na fiktivním klíči k *doteku*. To způsobí vymazání veškeré cache, která na tomto fiktivním klíči závisela. Pomocí tohoto mechanismu je možné zajistit aktuálnost dat [54].

U tabulky 4.1 je vidět, že fiktivní klíče jsou textové řetězce se specifickým formátem.

Tabulka 4.1: Ukázka fiktivních klíčů

Typ objektu	Formát fiktivního klíče	Příklad
Stránka	node|<site name>|<alias path>|<culture>	node|samplesite|/home|en-us
Objekt	<object type>|byid|<id>	cms.user|byid|53
Nastavení	cms.settingskey|<settings>	cms.settingskey|pageprefix

4.3 Typy cache

4.3.1 Cachování dat

Provádí se přímo v aplikační kódu prostřednictvím cachovací API poskytnuté Kenticem a slouží ke cachování dat během procesu generování výstupu stránky. Toto API je implementované na základech

aplikační cache ASP.NET. Vývojáři je umožněno přistupovat k tomuto mechanismu pomocí dvou tříd.

CacheHelper

Jedná se o třídu, prostřednictvím které je možné ukládat objekty do cache a následně k nim přistupovat. K tomu nabízí několik metod, které se hodí při práci se závislostními klíči [55]:

- `Cache<T>(Func<T> loadMethod, CacheSettings settings)` – načte data pomocí delegáta `loadMethod` a výsledek uloží do cache.
- `TouchKey(string key)` – vyvolá *dotek* fiktivního klíče a zneplatní všechny záznamy na něm závislé. Volá se automaticky během vytvoření či změny objektu v systému.
- `GetCacheDependency(string keys)` – převádí textový řetězec na objekt typu `CMSCacheDependency`, který se nastavuje objektu typu `CacheSettings`
- `EnsureDummyKey(string key)` – zaregistruje fiktivní klíč

V následující ukázce je vidět, jak se dá cachovat výsledek operace získání všech uživatelů v systému s využitím `CacheHelper` třídy [55]:

```
ObjectQuery<UserInfo> data = CacheHelper.Cache(
   cs => LoadUsers(cs), new CacheSettings(10, "
   customdatasource|users"));
```

```
// Metoda pro načtení dat
// Je zavolána pouze v případě, kdy její
// výsledek není uložen v mezipaměti.
private ObjectQuery<UserInfo> LoadUsers(
   CacheSettings cs)
{
    ObjectQuery<UserInfo> result = UserInfo.
       Provider.Get();

    // Nastavení závislostí
    cs.CacheDependency = CacheHelper.
       GetCacheDependency("cms.user|all");

    return result;
}
```

IProgressiveCache

Oproti předchozí třídě, instanci `IProgressiveCache` je možné získat podle návrhového vzoru vkládání závislostí [56]. Toto rozhraní navíc podporuje tzv. progresivní načítání, které zajišťuje, že pokud více asynchronních vláken načítá stejná data, operace načtení se provede pouze jednou, její výsledek se uloží do mezipaměti a ten je pak propagován ostatním čekajícím vláknům [55].

Oba přístupy se ve svých implementacích navzájem používají a v zásadě fungují stejně – obě třídy během ukládání záznamů do cache používají výchozí implementaci rozhraní `IMemoryCache` (3.2.1).

4.3.2 Rozšíření cachování v Razor pohledech

Rozšíření vychází z funkce cachování v Razor pohledech ASP.NET Core (3.2.3). V následující ukázce se obsah bloku, který zobrazuje seznam článků, ukládá do cache na jednu hodinu. Nevýhodou takového řešení je, že v případě úpravy některého z článků či přidání nového, se změna na výstupu neprojeví okamžitě.

```
<cache expires-after="@TimeSpan.FromHours(1)">
  @foreach (var article in @Model.Articles)
  {
    // ...
  }
</cache>
```

Od verze Kentico Xperience byl z toho důvodu přidán nový tag helper `cache-dependency`, pomoci kterého je možné definovat závislé objekty [57].

```
@{
  var deps =
    new[] { "nodes|mysite|article.type|all" };
}

<cache expires-after="@TimeSpan.FromHours(1)">
  <cache-dependency cache-keys="@deps" />
  @foreach (var article in @Model.Articles)
  {
    // ...
  }
</cache>
```

Závislost na fiktivním klíči `nodes|mysite|article.type|all` zajišťuje, že v případě změny některého z článků, se tato cache zneplatní. Nicméně jelikož toto rozšíření pracuje s rozhraním `IMemoryCache`, nelze jej použít v kombinaci s `DistributedCacheTagHelper` popsaný v obecné kapitole o ASP.NET Core cachovacích mechanismech (3.2.3).

4.3.3 Cachování souborů

Kentico podporuje nastavení cachování souborů jak na straně klienta, tak na straně serveru [58]:

- **na straně klienta** – Tento mechanismus přímo v prohlížeči duplikuje dříve požadované soubory a ty pak mohou být při násle-

dujícím požadavku vráceny uživateli bez přístupu k webovému serveru. Je toho dosaženo úpravou HTTP hlavičky odpovědi. Kentico umožňuje nastavit expiraci cache a nutnost její opětovné validace v případě vypršení platnosti. Nutno zmínit, že Kentico spravuje cache pouze souborů, které jsou obsluhovány systémovými HTTP zpracovateli (z anglického *HTTP handlers*).

- **na straně serveru** – V Kenticu je možné soubory ukládat buď do souborového systému nebo přímo do databáze. Pokud je soubor uložen do databáze, jeho cache v sobě drží veškerá binární data. V případě, že je soubor uložen v souborovém systému, jeho cache obsahuje jen základní informace typu metadat a binární data souboru jsou přenesena přímo ze souborového systému. V nastavení Kentica je možné nastavit expiraci cache a maximální velikost souborů, které se budou cachovat automaticky.

4.4 Synchronizace cache v rámci webové farmy

Jakmile v systému dojde k události, která vyvolá dotekovou operaci, budou z cache odstraněny závislé objekty pouze u serveru, na kterém ke změně došlo. Tento problém Xperience řeší využitím synchronizačních úloh (4.1.2). Při vyvolání operace dotyku se pro všechny ostatní servery webové farmy vytvoří synchronizační úloha typu `TouchCacheItemWebFarmTask`, která serveru přikazuje zavolání dotekové operace na zadaném klíči. Tím se zajistí odstranění neaktuálních dat v rámci celé webové farmy [59].

5 Požadavky na cache modul

Na základě předcházejícího rozboru rysů distribuované mezipaměti, aplikační cache technologie ASP.NET Core a práce s cache na platformě Kentico Xperience mohu sestavit seznam požadavků pro modul integrující distribuovaný cache systém do platformy Kentico Xperience.

5.1 Funkční požadavky

5.1.1 Integrace do Kentica Xperience

Funkce distribuované cache by měla být do platformy Kentico Xperience zařazena prostřednictvím sady vlastních rozhraní, které bude moci vývojář použít dle potřeby.

5.1.2 Závislosti záznamů

Stěžejní funkcí v současném cache mechanismu je funkce automatického zneplatnění cache při změně závislých dat. Je toho dosaženo pomocí tzv. fiktivních klíčů (4.2). Nový modul musí s tímto počítat a tuto funkci plně podporovat.

5.1.3 Priority záznamů pro uvolňování paměti

Ve spojitosti se zásady uvolňování (1.1.3) se v prostředí Kentico Xperience API můžou nastavit priority záznamů, podle kterých se vybírají kandidáti pro uvolnění z cache. Nový modul má brát úrovně těchto priorit v potaz a využívat je.

5.1.4 Metody po vymazání záznamu z cache

Cachování v ASP.NET Core podporuje nastavení metody, která se má vykonat po vymazání záznamu z cache (3.2.1). S rozhraním distribuované cache má také být možné jednotlivým záznamům nastavit akce po smazání.

5.1.5 Zachování expirace záznamů

V Kentico Xperience se používá absolutní a klouzavý typ expirace cache. Vytvářený modul distribuované cache musí oba typy expirace podporovat.

5.1.6 Synchronizace cache v rámci web farmy

Dle 4.4, servery Kentico Xperience aplikace si posílají synchronizační zprávy propagující změny v cachi za účelem uchování konzistentní verze lokální cache. V případě nově vytvářeného modulu, cache úložiště bude sjednocené pro všechny servery aplikace. Synchronizace cache mezi jednotlivými uzly webové farmy v případě distribuované cache tedy nebude potřeba.

5.1.7 Administrativní rozhraní pro správu cache

Kentico Xperience v rozhraní administrace nabízí ve spojení s cachí několik nástrojů určených ke správě a ladění. Mezi tyto nástroje patří výpis uložených záznamů, vymazání celého úložiště cache a monitorování operací na cache systému [60]. S nový modulem v administraci vznikne samostatné rozšíření ve formě aplikace pro správu distribuované cache. Tato aplikace musí pokrývat funkcionalitu zobrazení všech položek v cachi a možnost vyčištění celé distribuované cache.

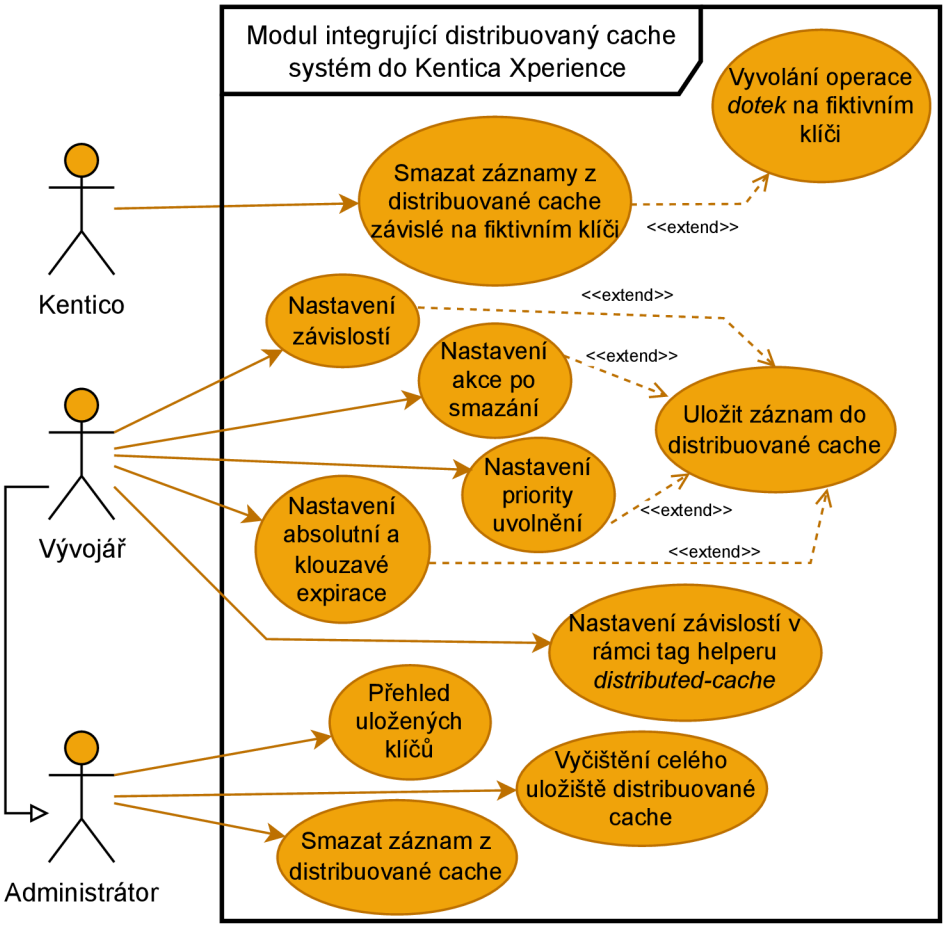
5.1.8 Trvalost cache

V současné době při restartu Kentico instance dojde ke kompletní ztrátě cache. U nového modulu je požadováno, aby běh distribuované cache nebyl spojen s během samotné Kentico aplikace a fungoval samostatně. Díky oddělení cachovací služby od zbytku aplikace bude tato podmínky splněna.

Trvalost cache ale způsobuje jiný problém v případě nasazení nové verze aplikace, kdy může dojít k nekonzistenci mezi novými a již nepodporovanými strukturami uložených v mezipaměti. To potvrzuje nutnost funkce manuální vymazání cache prostřednictvím administrace (5.1.7).

5.1.9 Podpora závislostních klíčů při cachování v Razor pohledech

K tag helperu `distributed-cache` v současné době neexistuje podpora závislostních klíčů platformy Kentico Xperience (4.3.2). Nový modul by měl obsahovat takové rozšíření, které v rámci tohoto tag helperu dovolí nastavit závislosti v podobě fiktivních klíčů aplikace Kentico.



Obrázek 5.1: Diagram případů užití

5.2 Nefunkční požadavky

5.2.1 Škálovatelnost

Vysoká míra nabízené škálovatelnosti je často podstatou distribuovaných systémů. V případě potřeby by mělo být možné navýšit kapacitu cache a celkový výkon. Za účelem zvýšení spolehlivosti cache systému by cachovací systém měl podporovat topologii replikace dat 2.3.1 a alespoň základní formu zálohy cachovaných záznamů na vnější paměťového úložiště.

5.2.2 Zabezpečení

Síťový přenos dat mezi webovým serverem a externím cache systémem musí být důvěryhodný a musí zaručovat integritu. Komunikace mezi Kentico aplikací a distribuovaných cache systémem by proto měla být šifrována.

5.2.3 Snadná integrace do existujícího systému

Integrace nového modulu do existujícího projektu by měla být co nejjednodušší. V ideálním případě by mělo k zaregistrování služby distribuované cache stačit rozšíření stávajícího aplikačního kódu bez nutnosti změny.

5.2.4 Dodržení principu oddělení zodpovědnosti

Modul by měl být navrhnut a implementován tak, aby respektoval princip oddělení zodpovědnosti a aby rozšíření funkcí modulu bylo pro jiného vývojáře co nejjednodušší. Příkladem takového rozšíření je například přidání podpory jiného systému distribuované cache nebo implementace vlastních metod pro serializaci a deserializaci dat.

6 Analýza cache systémů

V této kapitole provedu analýzu několika systémů, které mohou být použity pro distribuované cachování. Zabývat se budu jen těmi systémy, ke kterým existují knihovny pro integraci s technologií .NET a které jsou v současnosti stále aktuální. Každý systém stručně popíšu a srovnám s ostatními. Při výběru vhodného systému pro vytvářený modul budu nahlížet na seznam požadavků sepsaný v minulé kapitole.

6.1 Memcached

Memcached je mnohoúčelový open source systém distribuované paměti. Nejčastěji je využíván pro cachování objektů do paměti typu `RAM`. Původně byl vyvinut Bradem Fitzpatrickem pro LiveJournal[1] již v roce 2003 [61]. V době psaní této práce, poslední vydaná verze Memcached byla `1.6.22`.

Podporované platformy
Je nabízený zadarmo a je distribuován na operačních systémech typu Unix-Like (Linux, iOS a další) [62]. Existují i knihovny třetích stran, které podporují spuštění systému Memcached na operačním systému Microsoft Windows, nicméně se nejedná o oficiálně podporovaný přístup. Na cloudových platformách je Memcached nabízený jako plně spravovaná služba na Amazon Web Services nebo Google Cloud Platform [63, 64]. Naopak na platformě Microsoft Azure plně spravovaná verze Memcached v tuto chvíli není.

Architektura
Hlavní filozofií Memcached je být odlehčenou variantou systémů nabízejících distribuované cachování. Servery Memcached o sobě navzájem nevědí. Nedochází k žádné komunikaci ani synchronizaci. Z toho důvodu zde není možné nastavit replikaci uzlů. Přidáváním serverů se zvyšuje celková dostupná kapacita cache úložiště. Velká část logiky je přenesena na stranu klienta. Klient pomocí hashovacího

1. Ruská sociální síť, kde si uživatelé mohou zakládat své blogy či deníky.

algoritmu dokáže pomocí klíče zjistit, na kterém konkrétním uzlu se hledaná data mohou vyskytovat. Díky tomu klient provádí mazání a přepisování dat přímo na serveru, na kterém se data vyskytují [62].

Datové typy
Memcached ukládá dvojice klíč-hodnota jako řetězce a maximální velikost hodnoty je omezena na 1 MB. Jiné datové struktury Memcached v tuto chvíli nepodporuje [65].

Vícevláknové zpracování dotazů
Při zpracování příchozích operací dokáže Memcached server využít více vláken zároveň. Mechanismus zpracování více vláken je bezpečný (z anglického *thread-safe*) [66].

Zásady uvolňování a expirace
K uvolnění místa jsou strategie Memcached omezeny jen na algoritmus `LRU`. Nastavit zacházení s jednotlivými záznamy pomocí priorit zde podporováno není (1.1.3). Expiraci záznamu lze nastavit pouze absolutním časem [67, 65].

Nastavení závislostí
Memcached nepodporuje žádný způsob, jak nastavit závislosti mezi záznamy v cachi.

Sledování změn
V nativní verzi Memcached není podporován způsob, jak sledovat události (typu odstranění, změna) na záznamech.

Zabezpečení
Memcached podporuje autentizaci a šifrování komunikace přes `TLS` [68].

Persistence
Od verze `1.5.18`, Memcached obsahuje funkci *restart za tepla* (z anglického *warm restart*). Pomocí této funkce je možné nakonfigurovat Memcached způsobem, aby uchovával data na disk. Díky tomu je možné obnovit data v cachi i po restartu serveru [69].

6.2 Redis

Remote Dictionary Server (lze přeložit jako *vzdálený slovníkový server*), zkráceně Redis, je open source in-memory úložiště datových struktur, které se používá jako databáze, mezipaměť, zprostředkovatel zpráv (z anglického *message broker*) a streamovací systém. Redis byl vytvořen roku 2009 Salvatorem Sanfilippem [70]. Analýzu provádím na verzi `7.2.2`.

Podporované platformy
Stejně jako v případě Memcached, Redis je vyvíjen pro operační systémy typu Unix-Like. Windows oficiálně podporován není. Redis podporuje i kontejnerové platformy jako je Docker [71, 72]. Přední cloudové platformy Amazon Web Services, Microsoft Azure a Google Cloud Platform nabízí Redis jako plně spravovanou službu.

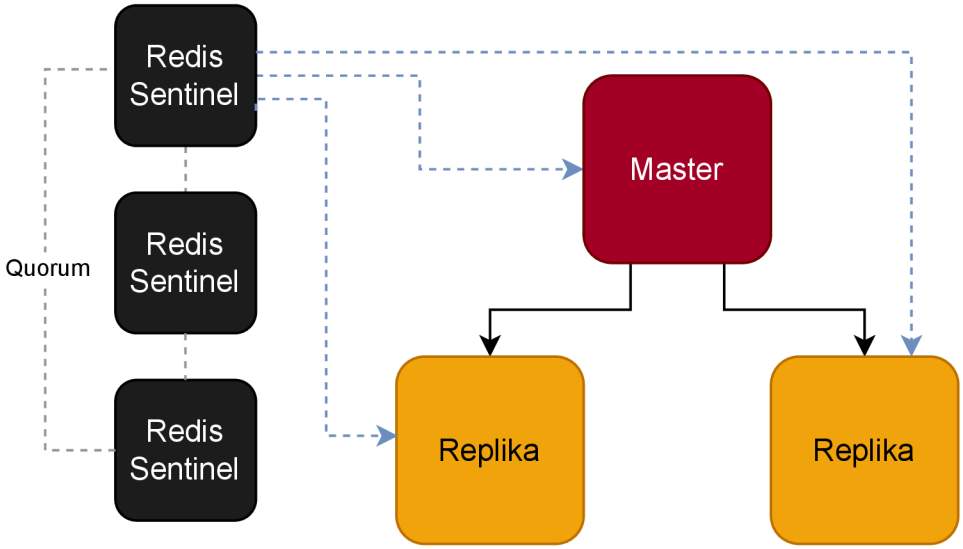
Datové typy
Redis poskytuje celou řadu datových struktur jako jsou řetězce, hashe, seznamy, množiny, setříděné množiny s dotazy na rozsah, bitové mapy a proudy (streamy). Velikost klíče a hodnoty je omezena na 512 MB [73].

Architektura
S Redisem je možné dosáhnout vysoké škálovatelnosti, dostupnosti a odolnosti vůči chybám pomocí pokročilých architektonických struktur. Vzor, který rozděluje data do několik uzlů se v nazývá **Redis Cluster**. Pro mapování klíčů na jednotlivé uzly, Redis používá strategii pojmenovanou Algoritmické třídění (z anglického *Algorithmic sharding*). Při té rozděluje data do slotů, kterých je vždy 16 386. Každý uzel je pak zodpovědný za podmnožinu těchto slotů. Vysoké dostupnosti lze dosáhnout replikací uzlů [26].

Redis Sentinel představuje jinou strategii k zajištění vysoké míry dostupnosti. Jedná se o distribuovaný systém procesů (tzv. sentinelů), které mezi sebou komunikují a navzájem kontrolují správné fungování všech Redis instancí – primárních uzlů a jejich replik. Redis Sentinel používá přístup založený na kvoru – minimální počet hlasů, které musí distribuovaná operace získat, aby mohla být provedena. V případě,

že má tedy dojít k přepojení některé z replik na primární uzel, musí k tomu udělit souhlas většina sentinelů [74, 75].

Na diagramu 6.1 lze vidět příklad architektury založené na technologii Redis Sentinel. Je zde jeden primární uzel, dvě jeho repliky a tři sentinel procesy. Pokud jeden ze sentinelů odhalí chybu u primárního uzlu, je potřeba ještě nejméně jednoho souhlasu od zbylých sentinel procesů, aby došlo k přepnutí jedné z replik na primární uzel.



Obrázek 6.1: Příklad Redis Sentinel [27]

Pro Redis je specifická jeho architektura s jedním vláknem. Znamená to, že požadavky klientů jsou zpracovávány jediným procesem. I přesto je, v porovnání s jinými systémy využívající architekturu více vláken, velmi výkonný. Použití jediného vlákna šetří spoustu času spotřebovaný na přepínání kontextu procesoru, není třeba zvažovat žádné operace zamykání a odemykání, které by mohly způsobit dodatečnou výkonnostní režii pro hlídání uvíznutí (*deadlock*) a nemůže zde dojít ani k chybě souběhu (*race condition*) [76].

Zásady uvolňování a expirace

Redis podporuje celou řadu strategií pro uvolnění paměti pro případ,

že byla dosažená maximální velikost úložiště. Například ke klasickým algoritmům `LRU` a `LFU` existují tzv. volatilní varianty (strategie `volatile-lru`, `volatile-lfu`). U volatilních strategií se berou v potaz jen záznamy s nastavenou expirací. Jinou možnou strategií je `no-eviction`, při které se žádné záznamy neuvolní a v případě zaplnění paměti jednoduše nepůjde nové záznamy přidat. Redis také podporuje strategie uvolňování založené na náhodném výběru. Nastavení priority záznamů v kontextu uvolňování z úložiště podporované není[77].

Redis u svých záznamů podporuje pouze absolutní typ expirace [78].

Sledování změn

Jak bylo zmíněno, Redis lze použít jako zprostředkovatele zpráv. To provádí vlastní implementací návrhového vzoru *publikování-odběr* (z anglického *Publish/Subscribe*). Tento vzor umožňuje jedné službě (*Publisher*) oznamovat asynchronně události několika příjemcům (*Subsribers*) bez přímé vazby odesílatelů s příjemci [79]. Pomocí této funkce se mohou vytvořit kanály, po kterých se budou zasílat události v systému Redis. Klienti následně mohou tyto kanály odebírat a být tak informování o událostech v cachi. Funkce *Redis Keyspace Notifications* sama obstarává publikování zpráv informujících o událostech na datech uložených v Redis úložišti. Nastavení konkrétních typů událostí je součástí konfigurace [80, 81].

Transakce

V Redisu je možné vytvářet transakce. Ty dovolují provést skupinu operací atomicky – v jednom kroku. Jiný příchozí požadavek nemůže být během provádění transakce obsloužen. Příkazy jsou prováděny jako jedna izolovaná operace a transakce se tedy provede celá nebo vůbec [82].

Nastavení závislostí

Stejně jako u Memcached, závislosti mezi záznamy v cache nastavovat nelze.

Persistence
Redis nabízí dva základní způsoby, jak ukládat data na trvalé úložiště typu pevný disk. Oba způsoby lze kombinovat [83]:

- `RDB` (Redis Database) – V pravidelných intervalech provádí snímky (z anglického *snapshots*) datového úložiště a ty pak ukládá na disk.
- `AOF` (Append Only File) – Zaznamenává každou operaci zápisu přijatou serverem. Tyto operace lze pak při spuštění serveru znovu přehrát a rekonstruovat tak stav původní datové sady.

Zabezpečení
Redis nativně podporuje řídit přístupy pomocí `ACL`. S touto funkcí je možné omezit určitá připojení z hlediska operací, které lze provádět, a klíčů, ke kterým lze přistupovat [84]. Od verze 6 Redis poskytuje `TLS` podporu pro všechny své komunikační kanály [85, 23].

6.3 Hazelcast

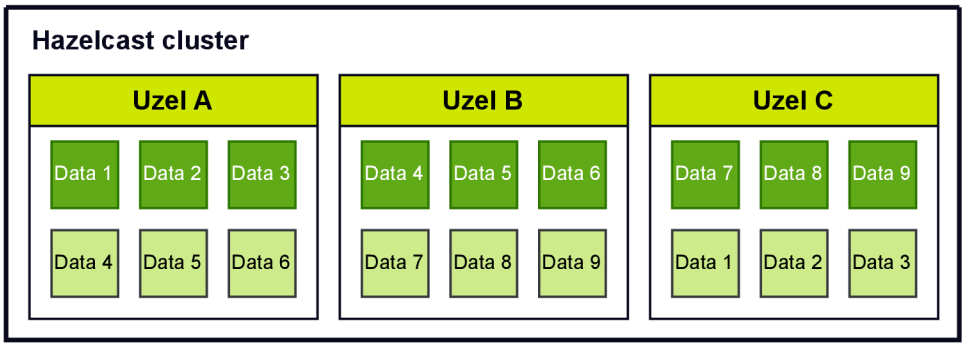
Hazelcast je open-source platforma určená pro ukládání distribuovaných datových struktur, distribuované výpočty a dá se použít jako zprostředkovatel zpráv. Byl vyvinut v programovacím jazyce Java v roce 2008 [86]. V době psaní této práce, poslední vydaná verze Hazelcastu je `5.3.2`.

Podporované platformy
Hazelcast poskytuje rozhraní API pro mnoho nejpoužívanějších programovacích jazyků, včetně jazyků Java, C++, C#, Python, Go a dalších. Lze jej také nasadit v několika různých cloudových prostředích jako Amazon Web Services, Google Cloud Platform nebo Microsoft Azure. Podporované je i nasazení Hazelcastu přes Kubernetes [87].

Architektura
Hazelcast lze nasadit na jednom počítači nebo v clusteru. Datové úložiště a výpočetní operace jsou rovnoměrně rozděleny mezi členy clusteru. Při přidání nového uzlu do clusteru dojde k automatického vybalancování zátěže [88].

Při inicializaci clusteru se jeho členové navzájem automaticky objeví a ustanoví mezi sebou TCP spojení, přes které pak spolu komunikují a sledují stav ostatních členů. Pokud se člen clusteru stane nedostupným (například v důsledku selhání sítě), ostatní členové spolupracují na diagnostice stavu a okamžitě převezmou odpovědnost za selhávajícího člena. K určení, zda je člen nedostupný, poskytuje Hazelcast vestavěné detektory selhání [89].

Hazelcast podporuje několik nástrojů pro lepší odolnost proti chybám – základem je schopnost replikování dat. To provádí přerozdělením záloh dat mezi členy clusteru. Počet záloh v clusteru je konfigurovatelný. Ve výchozím nastavení se počítá s jednou zálohou pro každý oddíl dat. Pokud u některého člena clusteru dojde k výpadku, Hazelcast přerozdělí zálohy na zbývající členy tak, aby pro každý oddíl existovala záloha [90]. Na obrázku 6.2 můžeme vidět přerozdělení dat a jejich záloh v rámci tříčlenného clusteru. Primární oddíly dat jsou vyznačené tmavě zelenou a jejich zálohy jsou vyznačené světle zelenou barvou.



Obrázek 6.2: Hazelcast replikace

Datové typy

Hlavním modelem ukládání dat Hazelcastu je úložiště dvojic klíč-hodnota v paměti typu in-memory. Hazelcast však podporuje i řadu dalších distribuovaných datových struktur odvozených z programovacího jazyka Java jako jsou Queue, Set, MultiMap, Topic a další [91].

Vícevláknové zpracování dotazů
Na rozdíl od služby Redis, Hazelcast ke zpracování požadavků využívá vícevláknovou architekturu. Všechny datové typy Hazelcastu jsou přizpůsobené pro zpracování více vlákny a operace na nich jsou proto *thread-safe*. V kontextu vícevláknového zpracování, Hazelcast dovoluje vysokou míru konfigurace [92].

Zásady uvolňování a expirace
Pro výběr záznamů pro smazání během uvolňování paměti při zaplnění se může použít algoritmus `LRU` nebo `LFU`. Priorita k záznamům přiřazovat nelze. Hazelcast podporuje nastavení absolutní i klouzavé expirace [93].

Sledování změn
Hazelcast obsahuje svoji vlastní implementaci Publish/Subscribe vzoru zvanou **Topic**. Z toho důvodu lze Hazelcast využít jako zprostředkovatele zpráv. Nicméně pro samotné sledování změn na datech uložených v Hazelcastu (ekvivalent funkce *Redis Keyspace Notifications*) neexistuje nativní podpora [94].

Zabezpečení
Komunikace s klientem a mezi uzly navzájem je plně šifrovaná pomocí TLS [95].

Nastavení závislostí
V současné verzi neexistuje nativní způsob, jak v Hazelcastu nastavit závislost mezi dvě záznamy v cache úložišti.

Persistence
V základní verzi Hazelcast nenabízí možnosti pro ukládání zálohy na trvalé úložiště. Nicméně ve verzi Enterprise Edition je možné pravidelné zálohy na disk nakonfigurovat [96]

6.4 Shrnutí analýzy

V této analýze jsem se zaměřil na nabízené funkce tří systémů distribuované cache.

Memcached se zaměřuje na rychlost a jednoduchost. Nenabízí pokročilé funkce, jako jsou transakce nebo replikace dat, a je vhodný pro jednoduché aplikace, které potřebují rychlý přístup k cachovaným datům.

Na druhou stranu, Redis a Hazelcast nabízí mnoho pokročilých funkcí. Podporují různé datové typy, transakce, Publish/Subscribe mechanismy a replikaci dat. Jsou vhodné pro širokou škálu aplikací, včetně těch, které vyžadují komplexní cachovaní a spolehlivost.

Klíčové vlastnosti lze porovnat v tabulce 6.1.

Tabulka 6.1: Srovnávací tabulka

Funkce/Vlastnost	Memcached	Redis	Hazelcast
Open source	✓	✓	✓
Plně spravováno na cloudu	AWS, GCP	AWS, Azure, GCP	AWS, Azure, GCP
TLS šifrování	✓	✓	✓
Různé datové typy	✗	✓	✓
Různé strategie uvolňování	✗	✓	✓
Nastavení priority uvolňování	✗	✗	✗
Expirace	Absolutní	Absolutní	Absolutní a klouzavá
Transakce	✗	✓	✓
Vytvoření perzistentní zálohy	✓	✓	✓
Publish/Subscribe	✗	✓	✓
Replikace	✗	✓	✓
Sharding	✓	✓	✓
Vícevláknové zpracování požadavků	✓	✗	✓
Maximální velikost klíče	250 bytů	512 MB	lze konfigurovat
Maximální velikost hodnoty	1 MB	512 MB	lze konfigurovat

6.5 Výběr vhodného systému

V minulé kapitole jsem definoval požadavky, které by měl vytvářený modul integrující funkcionalitu distribuované cache do platformy Kentico Xperience splňovat. Z vybraných systémů, pro které jsem provedl analýzu, žádný z nich ve svých nativních verzích nenaplňuje požadovanou funkcionalitu.

Některé funkce, jako závislosti mezi záznamy cache nebo nastavení priority při uvolnění z paměti, nepodporuje žádný systém. Z toho důvodu bude nutné tyto funkce implementovat dodatečně. K tomu bude pro jednodušší práci vhodné používat komplexnější datové typy, které dovolí vytvořit v distribuovaném úložišti vlastní pomocné struktury. Tím pádem se Redis a Hazelcast v porovnání se systémem Memcached jeví vhodněji.

Pro nastavení akce, která se má vykonat po smazání z cache, by se dobře dala využít funkce *Redis Keyspace Notifications*. U Hazelcastu by se tento požadavek plnil o něco náročněji.

Redis v porovnání se systémem Hazelcast poskytuje o něco lépe zpracovanou dokumentaci a disponuje větší komunitou vývojářů. V kontextu vývoje v technologii .NET je podstatná i skutečnost, že společnost Microsoft často integruje Redis do vlastních systémů. Z toho důvodu jsem se rozhodl ve své implementaci použít jako systém distribuované cache platformu **Redis**.

7 Implementace a kódová dokumentace

V této kapitole popíšu způsob implementace vlastního modulu pro integraci technologie distribuované cache do produktu Kentico Xperience. Modul jsem implementoval do vlastního projektu, který je možné distribuovat prostřednictvím NuGet balíčku. Výstupem této kapitoly je návod integrace tohoto modulu do existující projektu Kentico Xperience.

7.1 Použité technologie

Během vývoje jsem pracoval s celou řadou různých technologií. Samotný modul distribuované cache implementuje specifikaci NET Standard 2.0. Tato specifikace poskytuje jednotné API, které je možné použít na různých implementacích .NET, včetně .NET Framework a .NET Core [97]. Díky tomu může být modul použit jak v CMS, tak v livesite aplikaci. Při vývoji administrativního rozhraní jsem pracoval s technologií User Control[1]. Jako vývojové prostředí jsem použil Visual Studio. Pro použití instance Redis jsem se rozhodl využít cloudového řešení od Azure. Pro vývojové a testovací účely jsem modul implementoval v rámci demo projektu od Kentica. Pro lokální nasazení jsem využíval Správce Internetové informační služby[2] (z anglického *Internet Information Services*) a databázový Microsoft SQL Server. Pro účely testování funkcionalit jsem použil platformu pro práci s rozhraním API – Postman.

7.2 Analýza a návrh

7.2.1 Návrhový vzor cache mechanismu

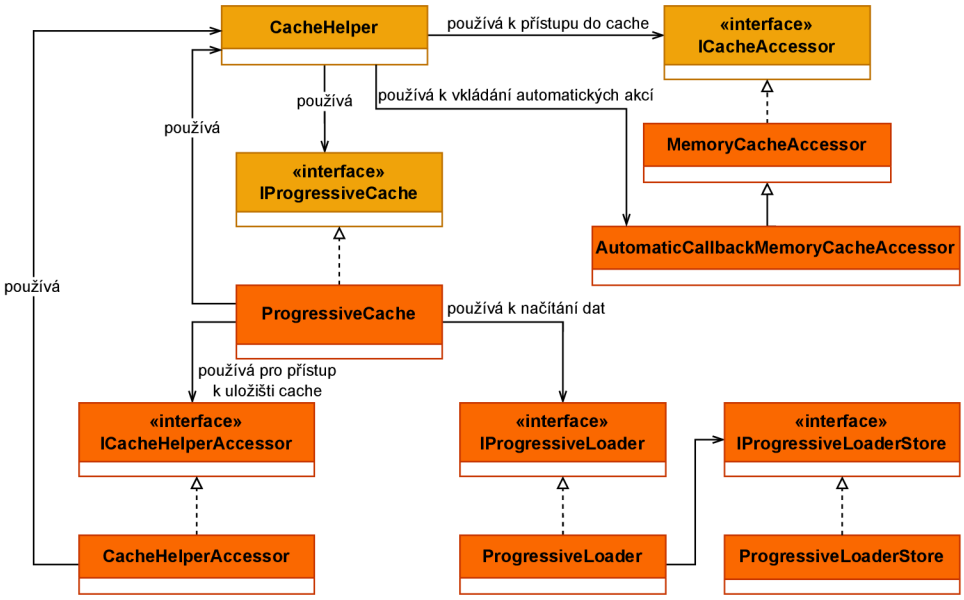
Současný cachovací mechanismus funguje na principu dotazování se cache na hledanou hodnotu. V případě nenalezení (cache-miss), jsou

1. V .NET Framework je User Control jedním z druhů vlastních ovládacích prvků, který je možné vytvořit pomocí technologie Windows Forms. User Control představuje znovupoužitelný prvek uživatelského rozhraní, který kombinuje existující ovládací prvky a může mít vlastní logiku a vlastnosti [98].
2. Softwarový webový server vytvořený společností Microsoft

data manuálně načtena a uložena do cache. Samotný systém cache s databází nijak nepracuje a ani se s ní samovolně nesynchronizuje. Podle rozboru v 1.3 je možné určit, že bude potřeba implementovat návrhový vzor `Cache-Aside`.

7.2.2 Analýza architektury nativního cachovacího mechanismu

Během návrhu struktury nového modulu jsem shledal za vhodné, zhodnotit strukturu současného in-memory cachovacího mechanismu platformy Kentico Xperience. Hlavní část jsem zaznamenal do diagramu tříd (7.1).



Obrázek 7.1: Struktura současného cachovacího mechanismu

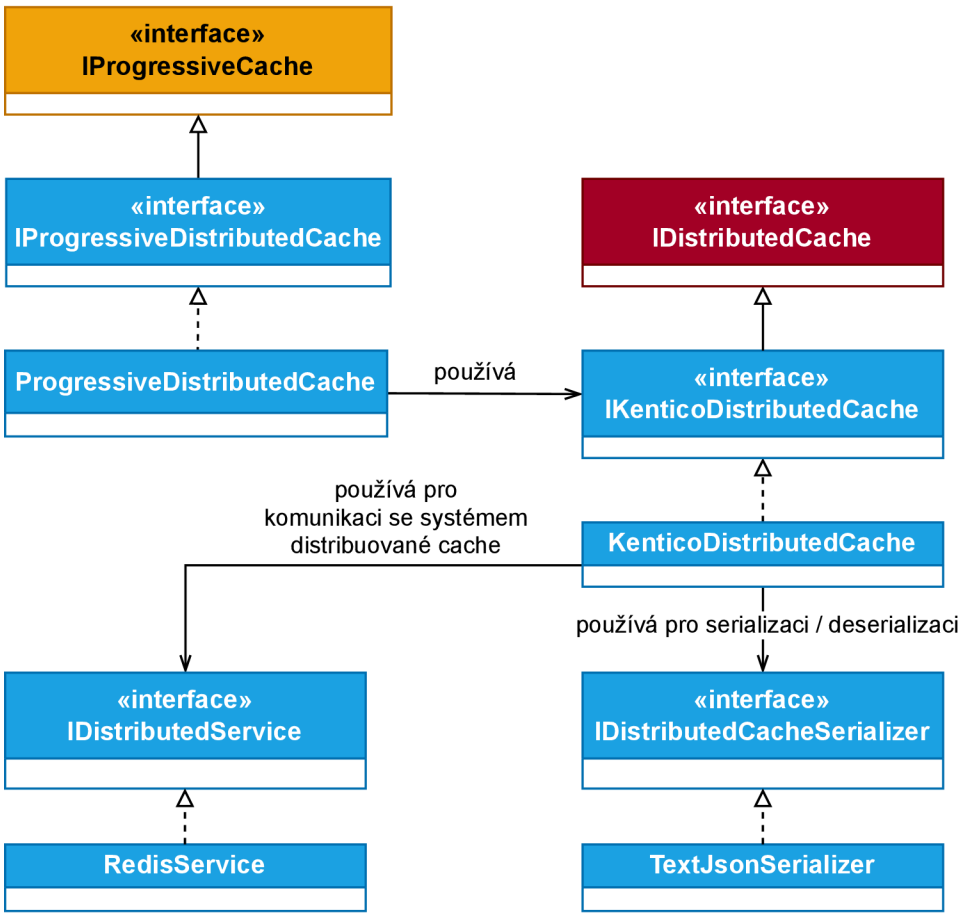
Lze si všimnou, že k uložení dat do cache se využívá implementace rozhraní `ICacheAccessor`. Nicméně třída `CacheHelper` mimo toto rozhraní ve zvláštních případech používá konkrétní implementaci `AutomaticCallbackMemoryCacheAccessor`, která rozšiřuje výchozí implementaci `ICacheAccessor` založenou na in-memory cachování ASP.NET. Takový návrh, obsahující těsné spojení (z anglického *tight coupling*),

nedovoluje nahrazení původního mechanismu vlastní implementací tak, aby byla zachována absolutní funkčnost.

7.2.3 Diagram tříd

Modul je navržen tak, aby jako veřejné API poskytoval implementace jak rozhraní `IProgressiveCache` od Kentica, tak i `IDistributedCache` od Microsoftu.

Při návrhu jsem se snažil vzít v úvahu princip oddělení zodpovědnosti. Především jsem oddělil logiku serializace od jádra implementace. To stejné jsem udělal pro funkce vázané na konkrétní distribuovaný systém. Na obrázku 7.2 je vidět kompletní návrh struktury. Diagram slouží především pro označení vztahů mezi třídami, proto zde nejsou zahrnuty metody a atributy tříd. Oranžovou barvou je označena již existující třída z knihovny Kentica. Červená barva zase značí třídu od Microsoftu.



Obrázek 7.2: Diagram tříd modulu distribuované cache

7.3 Implementační detaily služby pro práci s Redis

V této sekci zdokumentuji rozšiřující funkce pro práci s Redis. Veškeré tyto funkce jsou pro tuto platformu specifické a jsou implementované v rámci třídy `RedisService`.

7.3.1 Použité knihovny při implementaci

Pro práci s úložištěm platformy Redis jsem použil NuGet[3] balíček `StackExchange.Redis`. Ten poskytuje klienta pro práci s Redis úložištěm v prostředí .NET.

Dříve jsem zmínil, že Microsoft vytvořil vlastní implementaci rozhraní `IDistributedCache` pro Redis (3.2.2). Ta se nachází v balíčku `Microsoft.Extensions.Caching.StackExchangeRedis`. Po jeho analýze jsem se rozhodl tento NuGet ve své implementaci nepoužít. Možnosti přidání dodatečné funkcionality by s tímto balíčkem byly velmi omezené a mnohdy neefektivní – především z důvodu ztráty plné kontroly nad řízením transakčních operací při komunikaci s Redis instancí.

7.3.2 Funkce závislosti na fiktivním klíči

Návrh způsobu uchovávání cache závislostí

Jak bylo zmíněno v minulé kapitole, Redis přímo nepodporuje uchovávání závislostí mezi jednotlivými záznamy. Z toho důvodu je nutné implementovat vlastní systém uchovávání závislostí. Při návrhu jsem se nechal inspirovat implementací podobné funkce od Microsoftu, kde předmětem implementace byl mechanismus *cachování výstupu* (3.2.5) napojený právě na platformu Redis [46].

Finální návrh spočívá v rozdělení úložiště distribuované paměti na několik oddílů. Tyto oddíly jsou logicky odděleny podle předpony klíče záznamu. První oddíl je vyhrazen přímo pro cachované objekty (předpona `SDIPR_CK`). Účelem druhého oddílu je uložit informaci o závislostech na fiktivních klíčích (předpona `SDIPR_DCK`). Druhý oddíl obsahuje záznam `SDIPR_DCK` se všemi fiktivními klíči uložených v cachi a k tomu pro každý fiktivní klíč existuje záznam ve formátu `SDIPR_DCK_nazev-fiktivniho-klice` obsahující množinu na něm závislých klíčů.

Při takovém návrhu to znamená, že při doteku na fiktivní klíč stačí získat klíče ze záznamu `SDIPR_DCK_nazev-dotknuteho-klice` a ty následně z cache smazat.

3. Zkompilovaná knihovna funkcí v prostředí .NET podporující snadnou distribuci a správu verzí.

Tabulka 7.1: Návrh způsobu uchovávání závislostních klíčů

Klíč	Hodnota
SDIPR_CK_allArticlesCK	`ArticleModel[]`
SDIPR_CK_article25CK	`ArticleModel`

SDIPR_DCK	`[`
	`SDIPR_DCK_sdipr.article|all,`
	`SDIPR_DCK_sdipr.article|byid|25`
	`]`

SDIPR_DCK_sdipr.article|all	`[SDIPR_CK_allArticlesCK]`

SDIPR_DCK_sdipr.article|byid|25	`[`
	`SDIPR_CK_allArticlesCK,`
	`SDIPR_CK_article25CK`
	`]`

Implementace návrhu pro systém Redis

Pro uchovávání všech fiktivních klíčů a jejich závislých klíčů v systému Redis je použit datový typ setříděné množiny. Každému prvku množiny lze přiřadit skóre, podle kterého se můžeme dále na prvky množiny dotazovat [99].

Při vložení záznamu do Redis úložiště jsou všechny jeho fiktivní klíče uloženy do množiny `SDIPR_DCK`. Následně jsou množiny záznamů jednotlivých závislostních klíčů rozšířené o klíč vkládaného záznamu se skórem času expirace.

Platforma Redis nepodporuje nastavení expirace jednotlivým položkám množiny. Z toho důvodu po čase dochází k situacím, kdy záznam `SDIPR_DCK_nazev-klice` obsahuje klíče, které se již v cachi nevyskytují. Pro zmírnění této skutečnosti, je ve třídě `CleanRedisCache` vytvořena úloha, která se v pravidelných intervalech automaticky provádí. V prostředí Kentico Xperience se takové úloze říká *Scheduled task* [100]. Úloha prochází všechny fiktivní klíče z `SDIPR_DCK` a promazává z jejich množin již neaktuální klíče. Toho lze efektivně dosáhnout pomocí skóre reprezentující čas expirace daného klíče.

```
transaction.SortedSetRemoveRangeByScoreAsync(
    dependencyKey,      //název fiktivního klíče
    0,                  //od času 0
    DateTime.Now.Ticks  //do času současného
);
```

Rozšíření operace doteku fiktivního klíče

Pro zajištění zneplatnění neaktuálních cachovaných záznamů v rámci distribuované cache, je potřeba rozšířit operaci doteku na fiktivní klíč. Toho se dosáhlo zaregistrováním vlastní *helper* třídy `DistributedCacheHelper`. Ta rozšiřuje třídu `CacheHelper` a překrývá určité její metody.

```
[assembly: RegisterCustomHelper(typeof(
   DistributedCacheHelper))]
```

7.3.3 Priorita cache záznamu

Na platformě Redis se nedá nastavit priorita pro konkrétní záznamy, podle které by se k záznamu přistupovalo odlišně během procesu uvolnění úložiště.

Popis vlastního mechanismu

Pro zapojení funkce priorit v rámci služby Redis jsem implementoval mechanismus založený na uvolnění záznamů z cache ještě před úplným zaplnění úložiště. Tento mechanismus funguje jako úloha, která se automaticky spouští v pravidelných intervalech a je implementována v třídě `RedisPriorityBasedEviction`. V rámci úlohy se nejdříve zjistí aktuální stav zaplnění Redis úložiště, na jehož základě se smažou záznamy vybraných prioritních tříd.

V tabulce 7.2 jsou vypsané nastavené prahové hodnoty poměru zaplnění úložiště ke konkrétním prioritním třídám.

Tabulka 7.2: Nastavené prahové hodnoty

Práh	Prioritní třída
>= 0.85	`Low`
>= 0.9	`Normal`
>= 0.95	`High`

Nicméně stále může dojít k situaci, kdy se Redis úložiště zcela zaplní. V takovém případě situaci řeší přímo Redis. Ten prioritní třídy nebere v potaz a aplikuje vybranou strategii uvolňování, která se nastavuje v rámci konfigurace platformy (6.2).

Návrh způsobu uchovávání priority záznamu

Způsob uchovávání priorit je téměř totožný se způsobem ukládání závislostí na fiktivních klíčích. Úložiště obsahuje další logický oddíl s prefixem `SDIPR_PRIORITY`. Zde pro každou prioritu vzniká záznam ve formátu `SDIPR_PRIORITY_prioritni-trida`, která je opět datového typu setříděné množiny. Prvky množiny jsou klíče cachovaných záznamů s danou prioritou a jako skóre mají nastavený čas expirace záznamu. Díky tomuto návrhu je proces smazání záznamů s danou prioritou v úloze `RedisPriorityBasedEviction` vcelku efektivní.

Rozšíření úlohy pročištění cache

Stejně jako v případě uchovávání závislostí, i zde by po čase mohlo docházet k tomu, že záznamy typu `SDIPR_PRIORITY_prioritni-trida` budou obsahovat již expirované klíče. Z toho důvodu automatická úloha `CleanRedisCache` prochází i množiny jednotlivých prioritních tříd a podle skóre odebírá již expirované klíče.

7.3.4 Podpora akce po smazání

Již během rozhodování o tom, který systém distribuované cache ve svém modulu použiji, jsem uvažoval, že v případě Redisu by se možnost nastavení *akce po smazání* dala dobře implementovat s využitím funkce *Redis Keyspace Notifications*. Ta vychází ze zmiňovaného návrhového vzoru *Publish/Subscribe*, kde v tomto případě Redis publikuje

zprávy o změnách v cachi a strana webového serveru tyto zprávy odebírá. Aby Redis opravdu publikoval tyto zprávy, je potřeba nastavit proměnnou `notify-keyspace-events` [81].

```
> config set notify-keyspace-events KEA
```

Na straně aplikačního kódu jsem rozšířil metodu pro přidání nového záznamu do cache o nastavení *akce po smazání* zaregistrováním odběratele pro daný klíč. Po vykonání *akce po smazání* již není potřeba další zprávy z kanálu konkrétního klíče dále odebírat, a proto je odebírání kanálu zrušeno. Toho si lze všimnout v následující ukázce.

```
private void RegisterOnRemoveCallback(
  string key,
  //delegát akce po smazání
  PostEvictionDelegate onRemove
)
{
  _ = Subscriber.SubscribeAsync(RedisChannel.
     Pattern($"__keyspace@0__:{key}"), (channel,
     redisValue) =>
  {
    if (/* pokud záznam odebrán z~cache */)
    {
      //vykonej akci po smazání
      onRemove(/*...*/);
      //zruš odebírání tohoto kanálu
      Subscriber.Unsubscribe(RedisChannel.
         Pattern($"__keyspace@0__:{key}"));
    }
  });
}
```

7.3.5 Podpora klouzavé expirace

Redis přímo nepodporuje klouzavou expiraci. Bylo tedy potřeba tuto funkci implementovat v rámci aplikačního kódu. Hodnota klouzavé hodnoty je uložena přímo u hodnoty cachovaného objektu. Pokud je k objektu přistoupeno, je jeho expirace aktualizovaná. Při změně expirace záznamu je nutné aktualizovat skóre v závislostních a prioritních množinách.

7.4 Rozšíření tag helperu distribuované cache

Nově vzniklá třída `DistributedCacheDependencyTagHelper` představuje rozšíření existujícího tag helperu `distributed-cache`. Jeho implementace není závislá na konkrétní službě distribuované cache. K původnímu tag helperu přidává atribut `dependency-keys`, který přijímá pole závislostních klíčů v podobě textových řetězců.

```
@{
    var depKeys = new[] { "cms.user|all" };
}
<distributed-cache
    dependency-keys="@depKeys"
    name="my-distributed-cache-unique-key-103">
    Last update: @DateTime.Now.ToString("U")
</distributed-cache>
```

7.5 Serializace objektů

Metody serializace a deserializace objektů jsou oddělené od zbytku logiky modulu skrze rozhraní `IDistributedCacheSerializer`. Výchozí implementace serializuje data do formátu JSON[4] s využitím balíčku `System.Text.Json`. Původně jsem se snažil vytvořit univerzální serializér, který by dobře fungoval s Kentico objekty. Nicméně kvůli

4. JSON (*JavaScript Object Notation*) je datový formát určený pro výměnu dat mezi serverem a klientem [101]

složitosti těchto objektů, se mi nepodařilo implementovat funkční řešení.

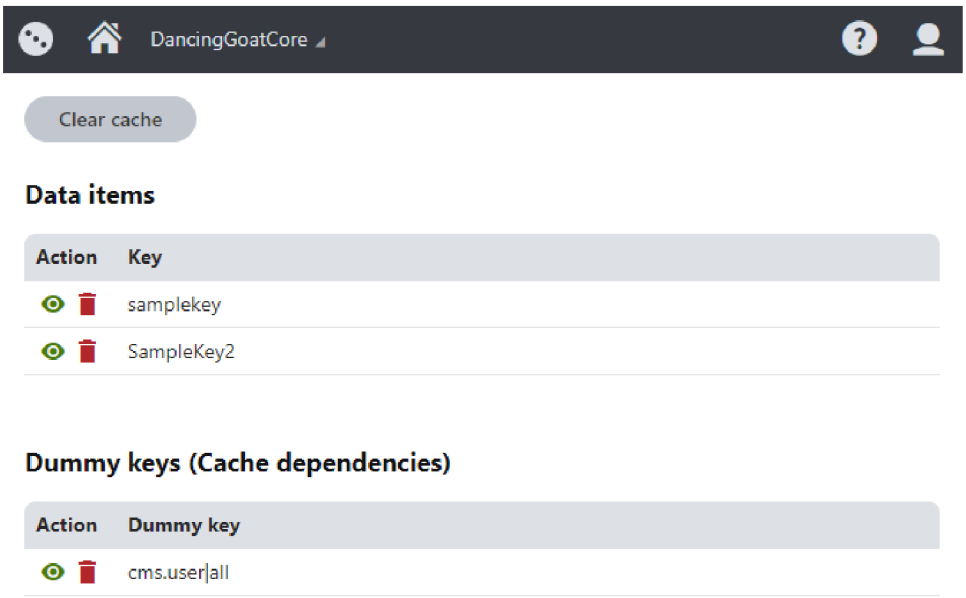
Implementací zmiňovaného rozhraní lze pro specifické potřeby vytvořit vlastní způsob serializace.

7.6 Administrační rozhraní

Pro správu distribuované cache v prostředí administrace systému jsem vytvořil vlastní Kentico modul. Ten do rozhraní administrace přidává aplikaci *Distributed Cache Overview*, která je svým rozvržením velmi podobná nativní aplikaci pro správu lokální cache. Aplikační rozhraní přehledu distribuované cache nabízí:

- možnost vyčistit celé úložiště distribuované cache
- přehled objektů a závislostních klíčů v cachi
- informace o času expirace a prioritní třídě jednotlivých záznamů
- možnost odebrání objektu z cache
- možnost odebrání závislostního klíče z cache – dojde k odstranění všech na něm závislých objektů

Jako zprostředkovatel komunikace mezi administrací a distribuovanou cachí slouží rozhraní `IDistributedCacheManager` jejíž implementace není závislá na konkrétní službě distribuované cache.



Obrázek 7.3: Rozhraní správy distribuované cache v administraci Kentica

7.7 Registrace služeb

Všechny potřebné služby se registrují při startu aplikace. Výchozí registraci služby distribuované cache jsem rozšířil o případ, kdy se nepodaří navázat se vzdálenou službou spojení. V takovém případě se zaregistruje implementace `NoCacheService`. Ta slouží jako záloha, aby v případě nedostupnosti systému distribuované cache byla aplikace stále funkční. Ačkoli požadavky na distribuovaný cache systém se budou ignorovat, tento stav se bude alespoň evidovat v protokolu událostí.

7.8 Metodika testování a ověření funkčnosti

Pro účely ověření funkčnosti jsem sestavil sadu koncových bodů, které jsem dotazoval prostřednictvím nástroje Postman. Jednotlivé mechanismy jsem testoval na skutečné instanci Kentico Xperience.

Jako Redis instanci jsem používal distribuci z cloudové platformy Microsoft Azure a lokální instalaci. Plně jsem otestoval i práci se službou Redis v replikované topologii. Předmětem testování nebyla rychlost vyřízení požadavků – ta závisí na jiných faktorech jako je stav sítě nebo, v prostředí cloudu, úroveň poskytnuté služby.

7.9 Integrace modulu do projektu

Pro zapojení modulu distribuované cache, pracující s platformou Redis, do existujícího projektu postupujte dle tohoto návodu:

1. Otevřete existující Kentico Xperience projekt ve Vašem vývojovém prostředí (např. Visual Studio).

2. K řešení přidejte projekt `Xperience.DistributedCache` a nastavte ho projektům CMS a livesite jako závislost.

3. Během inicializace CMS a livesite aplikace zaregistrujte služby modulu následovně:

```
using Xperience.DistributedCache;

serviceCollection.AddKenticoDistributedCache
    (
      Configuration //typu IConfiguration
);
```

4. Pro připojení k platformě Redis, nastavte u obou aplikací proměnnou prostředí `KenticoDistributedCacheConnectionString` řetězcem připojení k dané instanci.

5. Proměnou prostředí `KenticoDistributedCacheClearOnStartup` nastavte podle toho, zda si přejete vyčistit úložiště distribuované cache při startu aplikace.

6. Složku `CMSModules/DistributedCache`, která je součástí výstupu této práce, zkopírujte do CMS projektu.

7. Proveďte build celého řešení.
8. Spusťte administraci projektu.
9. V aplikaci *Sites* přes tlačítko *Import site or object* nahrajte soubor `distributed-cache-module.zip`
10. V aplikaci *Scheduled Tasks* zaregistrujte dvě nové úlohy:
    (a) Task display name: Clean Redis Cache
    Task provider assembly: `CleanRedisCache`
    Every: 1 Hour
    (b) Task display name: Redis Eviction
    Task provider assembly: `RedisPriorityBasedEviction`
    Every: 1 Hour

Závěr

Cílem práce bylo navrhnout a implementovat vlastní modul, který by integroval technologii distribuované cache paměti do platformy Kentico Xperience. Součástí procesu návrhu bylo zvolit konkrétního poskytovatele distribuované cache, kterého jsem vybral na základě analýzy a srovnání několika dostupných systémů tohoto typu.

Na začátku se práce zabývá představení technologie webové cache jako takové. Je vysvětlen základní koncept a problematika. Stručně jsou popsány úrovně, na kterých se cache může provozovat, a k tomu jsou prezentovány různé návrhové vzory a strategie, jak lze webovou cache implementovat.

Druhá kapitola je věnována technologii distribuované cache. V první části je srovnána s principem lokálního cachování. V obou případech jsou vysvětleny jejich koncepty a porovnány jejich silné a slabé stránky. Druhá část je věnována rozboru existující topologií distribuovaných cache systémů.

Třetí kapitola se věnuje možným realizacím popsaných konceptů v prostředí .NET.

V další kapitole jsou popsány funkce cachování v prostředí platformy Kentico Xperience. Nejdříve je platforma v krátkosti představena a následně jsou vysvětleny její specifické vlastnosti cachovacích mechanismů.

V následující kapitole je sestaven seznam požadavků, který je kladen na vznikající modul. Na základě tohoto seznamu, je provedena analýza a srovnání několika dostupných poskytovatelů distribuované cache. Ze systémů Memcached, Redis a Hazelcast byl jako nejvhodnější pro integraci vybrán druhý jmenovaný.

Poslední kapitola popisuje proces tvorby samotného modulu od návrhu architektury, s důrazem na možnou rozšiřitelnost, až po samotnou implementaci. Při návrhu struktury, která by plně nahradila stávající cache mechanismus, jsem narazil na limity v podobě neshovívavé a těžko rozšiřovatelné struktury stávajícího mechanismu a omezených možností serializace systémových objektů. I přes to výsledný modul splňuje všechny stanovené požadavky. V práci je kladena pozornost na zdokumentování implementačních detailů specifických pro propojení platforem Redis a Kentico Xperience.

Hlavním přínosem práce je vznik návrhu použití technologie distribuované cache ve spojení s platformou Kentico Xperience. Tato technologie zlepšuje možnosti škálování a obecně nabízí efektivnější správu webové cache. Na základě vypracované analýzy a implementace může být výstup práce využit ve společnosti BiQ Bluesoft a. s. k nahrazení lokálního přístupu cachování přístupem distribuovaným.

Modul distribuované cache lze v budoucnu rozšířit o podrobnější administrační rozhraní, prostřednictvím kterého administrátor systému získá více informací o konkrétních objektech uložených v distribuované cachi. Dále by modul mohl řešit případ výpadku na straně poskytovatele distribuované cache nastavením záložního plánu. V neposlední řadě se nabízí implementace mechanismu po vzoru *Blízká mezipaměť*, který by mohl vedle distribuovaného cache pracovat i s lokální cachí jednotlivých serverů ve webové farmě. Tím by se rapidně snížila latence způsobená serializací a síťovým přenosem.

A Elektronická příloha

Součástí práce je i elektronická příloha, která obsahuje:

- Zdrojové kódy: `Xperience_DistributedCacheModule.zip`

Bibliografie

1. LUTKEVICH, Ben. *cache* [online]. TechTarget, 2021-10 [cit. 2023-09-21]. Dostupné z: https://www.techtarget.com/searchstorage/definition/cache.
2. *Caching Overview* [online]. Amazon Web Services, 2023 [cit. 2023-09-21]. Dostupné z: https://aws.amazon.com/caching.
3. RICK ANDERSON John Luo, Steve Smith. *Cache in-memory in ASP.NET Core* [online]. Microsoft, 2023-04-11 [cit. 2023-09-13]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-7.0.
4. SINGH, Ela. *Caching Overview* [online]. Medium, 2021-10-01 [cit. 2023-09-21]. Dostupné z: https://medium.com/codex/caching-overview-cf9afaaedcff.
5. MCKINNON, Jenni. *What are Hit and Miss Ratios?* [online]. WP Rocket, 2023-05-10 [cit. 2023-09-23]. Dostupné z: https://wp-rocket.me/blog/calculate-hit-and-miss-ratios/.
6. *What is a cache hit ratio?* [online]. CloudFlare, 2023 [cit. 2023-09-22]. Dostupné z: https://www.cloudflare.com/learning/cdn/what-is-a-cache-hit-ratio/.
7. SANDU, Sudheer. *Distributed Caching — The Only Guide You'll Ever Need* [online]. Medium, 2022-07-08 [cit. 2023-09-20]. Dostupné z: https://medium.com/@sudheer.sandu/distributed-caching-the-only-guide-youll-ever-need-fe152357f912.
8. *Web Caching* [online]. Amazon Web Services, 2023 [cit. 2023-09-21]. Dostupné z: https://aws.amazon.com/caching/web-caching/.
9. *Cache-Control* [online]. Mozilla, 2023 [cit. 2023-09-22]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control.
10. *Content Delivery Network (CDN) Caching* [online]. Amazon Web Services, 2023 [cit. 2023-09-22]. Dostupné z: https://aws.amazon.com/caching/cdn/.

11. *What is a reverse proxy? | Proxy servers explained* [online]. CloudFlare, 2023 [cit. 2023-09-22]. Dostupné z: https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/.

12. *Caching patterns* [online]. Amazon Web Services, 2023 [cit. 2023-09-20]. Dostupné z: https://docs.aws.amazon.com/whitepapers/latest/database-caching-strategies-using-redis/caching-patterns.html.

13. *Cache-Aside pattern* [online]. Microsoft, 2023 [cit. 2023-09-20]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside.

14. FRANKEL, Nicolas. *A Hitchhiker's Guide to Caching Patterns* [online]. Hazelcast, 2023-12-07 [cit. 2023-09-21]. Dostupné z: https://hazelcast.com/blog/a-hitchhikers-guide-to-caching-patterns/.

15. MATT BRINKLEY, Jas Chhabra. *Caching challenges and strategies* [online]. Amazon Web Services, 2019 [cit. 2023-09-13]. Dostupné z: https://aws.amazon.com/builders-library/caching-challenges-and-strategies/.

16. GARCIA, Vinicius Fulber. *Caching: Cold Cache vs. Warm Cache* [online]. Baeldung, 2023-05-04 [cit. 2023-09-18]. Dostupné z: https://www.baeldung.com/cs/caching-cold-cache-vs-warm-cache.

17. NASIR, Mohsin. *Distributed caching in ASP.NET Core* [online]. Microsoft, 2023-07-28 [cit. 2023-09-14]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/performance/caching/distributed?view=aspnetcore-7.0.

18. *What is distributed caching?* [online]. redisson, 2022 [cit. 2023-09-19]. Dostupné z: https://redisson.org/glossary/distributed-caching.html.

19. PAUL KIRVAN, Stephen Bigelow. *What is a single point of failure (SPOF)?* [online]. TechTarget, 2021-11 [cit. 2023-09-20]. Dostupné z: https://www.techtarget.com/searchdatacenter/definition/Single-point-of-failure-SPOF.

20. TOM DYKSTRA, Genevieve Warren. *Serialization in .NET* [online]. Microsoft, 2023-04-05 [cit. 2023-09-25]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/standard/serialization/.

21. SALEEM, Muhammed. *Distributed Caching in ASP.NET Core* [online]. Microsoft, 2023-03-07 [cit. 2023-09-25]. Dostupné z: https://code-maze.com/aspnetcore-distributed-caching/.

22. *Co znamená zkratka ACL?* [online]. IT Slovník, 2020 [cit. 2023-09-25]. Dostupné z: https://it-slovnik.cz/pojem/acl.

23. *Redis security* [online]. Redis, 2023 [cit. 2023-09-25]. Dostupné z: https://redis.io/docs/management/security/.

24. *Security and Encryption in NCache* [online]. NCache by Alachisoft, 2022 [cit. 2023-09-25]. Dostupné z: https://www.alachisoft.com/ncache/security-encryption-features.html.

25. GABRIELSON, Jacob. *Avoiding fallback in distributed systems* [online]. Amazon Web Services, 2023 [cit. 2023-09-25]. Dostupné z: https://aws.amazon.com/builders-library/avoiding-fallback-in-distributed-systems/.

26. *Scale with Redis Cluster* [online]. Redis, 2023 [cit. 2023-09-25]. Dostupné z: https://redis.io/docs/management/scaling/.

27. YUSUF, Mahdi. *Redis Explained* [online]. Architecture Notes, 2022-08-11 [cit. 2023-09-25]. Dostupné z: https://architecturenotes.co/redis/.

28. *Cache Topologies* [online]. Ehcache, 2023 [cit. 2023-09-25]. Dostupné z: https://www.ehcache.org/documentation/2.8/get-started/cache-topologies.html.

29. *Caching Topologies for Linear Scalability* [online]. NCache by Alachisoft, 2022 [cit. 2023-09-25]. Dostupné z: https://www.alachisoft.com/ncache/caching-topology.html.

30. FRANKEL, Nicolas. *Near Cache* [online]. Hazelcast, 2022 [cit. 2023-09-21]. Dostupné z: https://docs.hazelcast.com/imdg/4.2/performance/near-cache.

31. BEŇOVSKÝ, David. *Developing Xperience applications using ASP.NET Core* [online]. Kentico Software, 2023-04-20 [cit. 2023-09-15]. Dostupné z: https://docs.xperience.io/developing-websites/developing-xperience-applications-using-asp-net-core.

32. BEŇOVSKÝ, David. *Xperience development models* [online]. Kentico Software, 2023-07-18 [cit. 2023-09-15]. Dostupné z: https://docs.xperience.io/developing-websites/xperience-development-models.

33. WARREN, Genevieve. *What is .NET? Introduction and overview* [online]. Microsoft, 2022-03-24 [cit. 2023-09-15]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/introduction.

34. *ASP.NET MVC Pattern* [online]. Microsoft, 2023 [cit. 2023-09-20]. Dostupné z: https://dotnet.microsoft.com/en-us/apps/aspnet/mvc.

35. RICK ANDERSON Dave Brock, Kirk Larkin. *Introduction to Razor Pages in ASP.NET Core* [online]. Microsoft, 2023-08-28 [cit. 2023-09-20]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-7.0&tabs=visual-studio.

36. RICK ANDERSON Luke Latham, Daniel Roth. *ASP.NET Core Blazor* [online]. Microsoft, 2023-09-08 [cit. 2023-09-20]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-7.0.

37. DYKSTRA, Tom. *Overview of ASP.NET Core* [online]. Microsoft, 2022-11-15 [cit. 2023-09-15]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/extensions/caching.

38. RICK ANDERSON, Kirk Larkin. *Overview of caching in ASP.NET Core* [online]. Microsoft, 2023-04-11 [cit. 2023-09-13]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/performance/caching/overview?view=aspnetcore-7.0.

39. *Caching in .NET* [online]. Microsoft, 2023-06-23 [cit. 2023-09-15]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/extensions/caching.

40. ANDERSON, Rick. *Memory management and garbage collection (GC) in ASP.NET Core* [online]. Microsoft, 2022-09-08 [cit. 2023-09-15]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/performance/memory?view=aspnetcore-7.0.

41. KELLNER, Peter. *Cache Tag Helper in ASP.NET Core MVC* [online]. Microsoft, 2022-06-03 [cit. 2023-08-05]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/cache-tag-helper?view=aspnetcore-7.0.

42. KELLNER, Peter. *Distributed Cache Tag Helper in ASP.NET Core* [online]. Microsoft, 2022-06-03 [cit. 2023-09-12]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/distributed-cache-tag-helper?view=aspnetcore-7.0.

43. RICK ANDERSON, Kirk Larkin. *Response caching in ASP.NET Core* [online]. Microsoft, 2023-07-27 [cit. 2023-03-17]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/performance/caching/response?view=aspnetcore-7.0.

44. DYKSTRA, Tom. *Output caching middleware in ASP.NET Core* [online]. Microsoft, 2023-07-27 [cit. 2023-09-14]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/performance/caching/output?view=aspnetcore-7.0.

45. DYKSTRA, Tom. *Output caching middleware in ASP.NET Core* [online]. Microsoft, 2023-07-27 [cit. 2023-09-14]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/performance/caching/output?view=aspnetcore-8.0.

46. *Implement output-cache store using redis* [online]. dotnet/aspnetcore, 2023 [cit. 2023-09-15]. Dostupné z: https://github.com/dotnet/aspnetcore/pull/48450.

47. BEŇOVSKÝ, David. *Upgrade from Kentico Xperience 13* [online]. Kentico Software, 2023-10-20 [cit. 2023-11-04]. Dostupné z: https://docs.xperience.io/xp/developers-and-admins/upgrade-from-kentico-xperience-13.

48. *What Is a Digital Experience Platform (DXP) | Kentico* [online]. Kentico Software, 2023 [cit. 2023-05-17]. Dostupné z: https://www.kentico.com/discover/what-is-a-dxp.

49. *Build Digital Experiences with .NET MVC | Kentico* [online]. Kentico Software, 2023 [cit. 2023-05-17]. Dostupné z: https://www.kentico.com/platforms/kentico-xperience-13/platform/mvc-development.

50. BEŇOVSKÝ, David. *Managing sites* [online]. Kentico Software, 2021-05-21 [cit. 2023-09-19]. Dostupné z: https://docs.xperience.io/configuring-xperience/managing-sites.

51. BEŇOVSKÝ, David. *Web farm internals and API* [online]. Kentico Software, 2018-11-25 [cit. 2023-09-20]. Dostupné z: https://docs.xperience.io/configuring-xperience/setting-up-web-farms/web-farm-internals-and-api.

52. BEŇOVSKÝ, David. *Setting up web farms* [online]. Kentico Software, 2021-05-18 [cit. 2023-09-19]. Dostupné z: https://docs.xperience.io/configuring-xperience/setting-up-web-farms.

53. BEŇOVSKÝ, David. *Configuring web farm servers* [online]. Kentico Software, 2023-01-11 [cit. 2023-09-19]. Dostupné z: https://docs.xperience.io/configuring-xperience/setting-up-web-farms/configuring-web-farm-servers.

54. BEŇOVSKÝ, David. *Setting cache dependencies* [online]. Kentico Software, 2022-01-24 [cit. 2023-05-10]. Dostupné z: https://docs.xperience.io/configuring-xperience/configuring-caching/setting-cache-dependencies.

55. BEŇOVSKÝ, David. *Caching in custom code* [online]. Kentico Software, 2021-03-30 [cit. 2023-05-10]. Dostupné z: https://docs.xperience.io/configuring-xperience/configuring-caching/caching-in-custom-code.

56. PINE, David. *.NET dependency injection* [online]. Microsoft, 2022-10-13 [cit. 2023-09-15]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection.

57. *Caching news: The cache-dependency tag* [online]. Kentico Software, 2021-04-07 [cit. 2023-08-05]. Dostupné z: https://devnet.kentico.com/articles/caching-news-the-cache-dependency-tag.

58. BEŇOVSKÝ, David. *Caching files and resources* [online]. Kentico Software, 2021-10-11 [cit. 2023-09-12]. Dostupné z: https://docs.xperience.io/configuring-xperience/configuring-caching/caching-files-and-resources.

59. BEŇOVSKÝ, David. *Debugging web farms* [online]. Kentico Software, 2020-10-15 [cit. 2023-09-20]. Dostupné z: https://docs.xperience.io/configuring-xperience/setting-up-web-farms/debugging-web-farms.

60. BEŇOVSKÝ, David. *Debugging the application cache* [online]. Kentico Software, 2021-03-16 [cit. 2023-09-27]. Dostupné z: https://docs.xperience.io/configuring-xperience/configuring-caching/debugging-the-application-cache.

61. MORALES, David. *What is Memcached?* [online]. Medium, 2021-01-28 [cit. 2023-09-20]. Dostupné z: https://medium.com/swlh/what-is-memcached-d1498623db3b.

62. *Overview* [online]. Memcached, 2020-10-26 [cit. 2023-09-28]. Dostupné z: https://github.com/memcached/memcached/wiki/Overview.

63. *Memorystore for Memcached* [online]. Google Cloud, 2023 [cit. 2023-09-28]. Dostupné z: https://cloud.google.com/memorystore/docs/memcached.

64. *Amazon ElastiCache for Memcached* [online]. Amazon Web Services, 2023 [cit. 2023-09-28]. Dostupné z: https://aws.amazon.com/elasticache/memcached/.

65. DUCKSBURY, Shane. *Redis™ vs Memcached* [online]. instaclustr, 2021-02-25 [cit. 2023-09-28]. Dostupné z: https://www.instaclustr.com/blog/redis-vs-memcached/.

66. *Comparing Redis and Memcached* [online]. Amazon Web Services, 2023 [cit. 2023-09-28]. Dostupné z: https://aws.amazon.com/elasticache/redis-vs-memcached/.

67. *What happens when Memcached is full?* [online]. DragonFly, 2023 [cit. 2023-09-28]. Dostupné z: https://www.dragonflydb.io/faq/what-happens-when-memcached-is-full.

68. *TLS* [online]. Memcached, 2019-09-30 [cit. 2023-09-28]. Dostupné z: https://github.com/memcached/memcached/wiki/TLS.

69. *WarmRestart* [online]. Memcached, 2022-02-18 [cit. 2023-09-28]. Dostupné z: https://github.com/memcached/memcached/wiki/WarmRestart.

70. *Introduction to Redis* [online]. Redis, 2023 [cit. 2023-09-28]. Dostupné z: https://redis.io/docs/about/.

71. *Supported platforms* [online]. Redis, 2023 [cit. 2023-09-29]. Dostupné z: https://docs.redis.com/latest/rs/installing-upgrading/install/plan-deployment/supported-platforms/.

72. *Install Redis on Windows* [online]. Redis, 2023 [cit. 2023-09-29]. Dostupné z: https://redis.io/docs/getting-started/installation/install-redis-on-windows/.

73. *Redis data types* [online]. Redis, 2023 [cit. 2023-09-29]. Dostupné z: https://redis.io/docs/data-types/.

74. FERNANDEZ, Tomas. *Understanding Redis High-Availability Architectures* [online]. semaphore, 2023-07-12 [cit. 2023-10-01]. Dostupné z: https://semaphoreci.com/blog/redis-architectures.

75. *High availability with Redis Sentinel* [online]. Redis, 2023 [cit. 2023-09-28]. Dostupné z: https://redis.io/docs/management/sentinel/.

76. *Diagnosing latency issues: Single threaded nature of Redis* [online]. Redis, 2023 [cit. 2023-09-30]. Dostupné z: https://redis.io/docs/management/optimization/latency/#single-threaded-nature-of-redis.

77. *Key eviction* [online]. Redis, 2023 [cit. 2023-09-28]. Dostupné z: https://redis.io/docs/reference/eviction/.

78. *EXPIRE* [online]. Redis, 2023 [cit. 2023-09-25]. Dostupné z: https://redis.io/commands/expire/.

79. BÜHNOVÁ, Barbora. Analytické a návrhové vzory a jejich aplikace [online]. 2013, s. 27 [cit. 2023-09-29]. Dostupné z: https://is.muni.cz/el/fi/jaro2018/PV167/um/Patterns-Bara.pdf.

80. *Redis Pub/Sub* [online]. Redis, 2023 [cit. 2023-09-28]. Dostupné z: https://redis.io/docs/interact/pubsub/.

81. *Redis keyspace notifications* [online]. Redis, 2023 [cit. 2023-09-29]. Dostupné z: https://redis.io/docs/manual/keyspace-notifications/.

82. *Transactions* [online]. Redis, 2023 [cit. 2023-09-28]. Dostupné z: https://redis.io/docs/interact/transactions/.

83. *Redis persistence* [online]. Redis, 2023 [cit. 2023-09-28]. Dostupné z: https://redis.io/docs/management/persistence/.

84. *ACL: Redis Access Control List* [online]. Redis, 2023 [cit. 2023-09-30]. Dostupné z: https://redis.io/docs/management/security/acl/.

85. *TLS: Redis TLS support* [online]. Redis, 2023 [cit. 2023-09-30]. Dostupné z: https://redis.io/docs/management/security/encryption/.

86. *Hazelcast Overview* [online]. Hazelcast, 2023 [cit. 2023-10-05]. Dostupné z: https://docs.hazelcast.com/hazelcast/5.3/.

87. *Public Clouds* [online]. Hazelcast, 2023 [cit. 2023-10-05]. Dostupné z: https://docs.hazelcast.com/hazelcast/5.3/deploy/deploying-in-cloud.

88. *Hazelcast Architecture* [online]. Hazelcast, 2023 [cit. 2023-10-01]. Dostupné z: https://docs.hazelcast.com/hazelcast/5.3/architecture/architecture.

89. *Discovery Mechanisms* [online]. Hazelcast, 2023 [cit. 2023-10-01]. Dostupné z: https://docs.hazelcast.com/hazelcast/5.3/clusters/discovery-mechanisms.

90. *Data Partitioning and Replication* [online]. Hazelcast, 2023 [cit. 2023-10-01]. Dostupné z: https://docs.hazelcast.com/hazelcast/5.3/architecture/data-partitioning.

91. *Distributed Data Structures* [online]. Hazelcast, 2023 [cit. 2023-10-01]. Dostupné z: https://docs.hazelcast.com/imdg/4.2/data-structures/distributed-data-structures.

92. *Threading Model* [online]. Hazelcast, 2023 [cit. 2023-10-03]. Dostupné z: https://docs.hazelcast.com/imdg/4.2/performance/threading-model.

93. *Managing Map Memory* [online]. Hazelcast, 2023 [cit. 2023-10-05]. Dostupné z: https://docs.hazelcast.com/hazelcast/5.3/data-structures/managing-map-memory.

94. *Topic* [online]. Hazelcast, 2023 [cit. 2023-10-03]. Dostupné z: https://docs.hazelcast.com/hazelcast/5.3/data-structures/topic.

95. *Security overview* [online]. Hazelcast, 2023 [cit. 2023-10-03]. Dostupné z: https://docs.hazelcast.com/hazelcast/5.3/security/overview#hide-nav.

96. *Persisting Data on Disk* [online]. Hazelcast, 2023 [cit. 2023-10-01]. Dostupné z: https://docs.hazelcast.com/hazelcast/5.3/storage/persistence.

97. WARREN, Genevieve. *.NET Standard* [online]. Microsoft, 2023-11-15 [cit. 2023-11-20]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/standard/net-standard?tabs=net-standard-2-0.

98. WARREN, Genevieve. *Create a .NET Framework Windows Forms user control that supports simple data binding* [online]. Microsoft, 2023-04-11 [cit. 2023-11-20]. Dostupné z: https://learn.microsoft.com/en-us/visualstudio/data-tools/create-a-windows-forms-user-control-that-supports-simple-data-binding?view=vs-2022&tabs=csharp.

99. *Redis sorted sets: Introduction to Redis sorted sets* [online]. Redis, 2023 [cit. 2023-10-25]. Dostupné z: https://redis.io/docs/data-types/sorted-sets/.

100. REBRO, Marcel. *Developing Xperience applications using ASP.NET Core* [online]. Kentico Software, 2023-03-23 [cit. 2023-10-15]. Dostupné z: https://docs.xperience.io/configuring-xperience/scheduling-tasks/scheduling-custom-tasks.

101. *What is JSON?: Store, query, and index JSON with Amazon DocumentDB* [online]. Amazon Web Services, 2023 [cit. 2023-10-28]. Dostupné z: `https://aws.amazon.com/documentdb/what-is-json/`.