NI-AM1

Ondřej Wrzecionko

ZS 2022/2023

Obsah

1 Úvod

1.1 Hodnocení

5 úkolů, co 2 týdny – minimálně 20b na zápočet, max. 40 bodů. Písemná zkouška ze tří části po 20 bodech.

1.2 Plán

Detaily komunikace, optimalizace výkonu v HTTP, reprezentace dat. Monolitická architektura (aplikační server) vs mikroslužby.

Cloudové služby (Software/Platform/Infrastructure as a service) → především integrace, aplikační server, ESB (enterprise service bus).

1.3 Middleware

Technologie se často mění, architektura se nemění zas tak často. Architektura přidává hodnotu do komunikačního systému a pomáhá při integraci aplikací. Jedná se o součást webu 2.0. Svět internetu jde rychle, ale nasazení do praxe bývá pomalejší.

Co může být middleware?

- identity access management (IdAM)
- messaging-oriented middleware (MOM) – *synchronní, asynchronní komunikace*
- protokol (např. SOAP, ale obecně cokoliv pro komunikaci)
- škálovatelnost, load-balancing (nginx)
- monitoring, logování dat, firewall
- stejně jako se snažíme o přepoužitelný kód, tak stejně se snažíme i o přepoužitelné služby
- chceme efektivní aplikace, šetřit náklady

2 Architektury informačních systémů

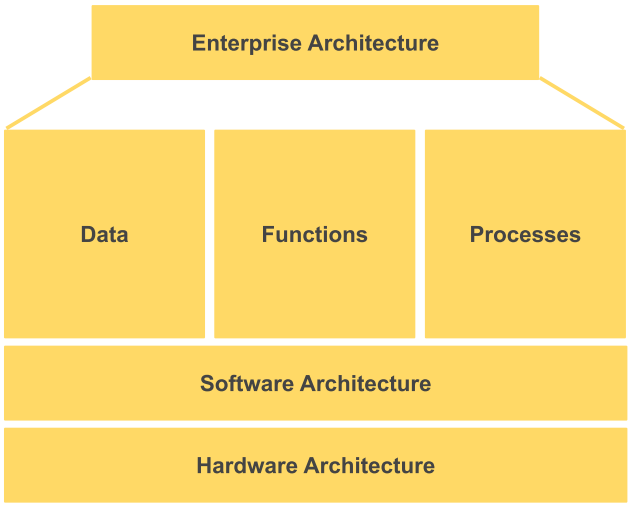
2.1 Přehled

Architektura informačního systému: popis struktury a chování systému

Mezi základní složky informačního systému patří:

- data, funkce, procesy (doménově závislé, *SaaS*)
- technické detaily:
  - software (*PaaS*)
  - hardware (*IaaS*)

Popis architektury provedeme pomocí **jednotlivých modelů** (datový model, hierarchie funkcí, activity diagram, stavový diagram). **Struktura** je určena daty a funkcemi, **chování** procesy.



Obrázek 1: Základní složky informačního systému

Vývojový koncept: návrh konceptu → obecný koncept → konkrétní implementace. To také závisí na **metodice** (analýza, návrh, implementace, testování, údržba) a **účastnících** (role: koncový uživatel, architekt, vývojář, administrátor).

Enterprise architektura se skládá z:

- EIS (executive information systems) – strategické rozhodování (např. dolování dat)

- BSS (business support systems) – systémy na úrovni organizace (např. oddělení financí)
- OSS (operational support systems) – nejnižší úroveň, (např. systém pro kontrolu překročení limitu dat v telekomunikační síti)

a boční systémy OIS (správa dokumentů) a B2B (komunikace mezi firmami).

Typy organizace:

- vendor *(poskytovatel)* – dodává aplikaci, produkt (product manager)
- supplier *(dodavatel)* – přizpůsobí produkt na míru (techničtí a solution architekti, adminové)
- customer *(uživatel)* – definuje požadavky

Role architektů:

- technický architekt – architektura na technické úrovni
- solution architekt – souvisí s dodavatelem, navrhuje řešení na konkrétní technologii
- enterprise architekt – na vysoké úrovni, znají architekturu aplikací na úrovni aplikační (procesy, datové modely)

2.2 Funkce a procesy

Klasifikace procesů:
→ Level 0 *(business funkce)* – správa zákazníků
→ Level 1 *(skupiny procesů)* – správa operací zákazníků
→ Level 2 *(core business procesy)* – správa objednávek
→ Level 3 *(business aktivity)* – vytvoření objednávky, uzavření objednávky
→ Level 4 *(business task)* – vyúčtování objednávky
→ Level 5 *(business step)* …

Obrázek 2: Order to Cash proces u telekomunikačního operátora

2.3 Data

Syntaxe: datový formát, reprezentace, serializace
→ XML, JSON, objektivní datové modely, SQL, YAML …
→ musíme provést validaci, máme zde ale nástroje (XSL, schémata)

Sémantika: význam dat v rámci domény, ve které jsou použity, chápeme ji jak přirozeným jazykem, tak skrze data a strukturu.

2.4 Integrace aplikací

Existuje integrace vnitropodniková (SOA) a integrace mezi podniky (B2B). Klíčem k integraci je **rozhraní**, to se opět skládá z:

- data – vstup, výstup, struktury pro popis chybových zpráv
- funkce – operace
- procesy – veřejný proces *("návod k použití", stavový diagram)*
- technické aspekty – IP adresa, protokol

2.5 Softwarové architektury

Existuje více úrovní, na kterých rozhraní fungují:
→ DBMS (data, funkce, procesy)
→ JDBC
→ JDO *(Java třídy na data objekty)*
→ Domain Object Model

Separation of Concerns

Při vytváření aplikace můžu mít více vrstev (klient, server; nebo dvě rozdílné komponenty), je třeba se domluvit na rozhraní mezi jednotlivými vrstvami *(manipulace s daty, integrita dat)*, jednotlivé vrstvy musí být nezávislé.

Client-server

- centrální server, skupina klientů
- monolitická / dvojvrstvá / třívrstvá …
- **single point of failure** = pokud server vypadne, máme problém
  → proto zavedeny ochranné mechanismy

Peer-to-Peer

- spolehlivý (když uzel selže, jiné uzly zaberou jeho funkci)
- škálovatelný (více uzlů sdílí zátěž – messaging systémy v enterprise systémech)

Existuje více typů architektur:

Monolitická

Všechny vrstvy na jednom počítači, nepřesnosné aplikace na specifický operační systém. Samostatné aplikace, minimální integrace, lokální úložiště dat.

Nevýhodou je složitá údržba a problémy s integritou dat, škálovatelností a výkonem.

Dvojvrstvá

- prezentační a aplikační vrstva jsou odděleny od datové vrstvy
- datová vrstva je na specifickém serveru
- více uživatelů, sdílejících databázi
- SQL + nativní aplikace

Nevýhodou je thick klient, který je náročný na údržbu a integrace na data.

Třívrstvá

Každá vrstva je na specifickém zařízení (tenký klient, komunikuje přes protokol se serverem, ten komunikuje s databází). Ani to ale nestačí …

Middleware

Přidáváme další middleware vrstvy mezi klienta a server. Problémem je nutnost škálovat jako celek, proto přišla:

Architektura mikroslužeb

Jednotlivé middlewary, aplikace a DB monolity se rozpadají na jednotlivé mikroslužby, které jde již dobře škálovat.

2.6 Typy middlewarů

Škálovatelnost

Zajišťují škálovatelnost, messaging servery, load balancers, proxy servery, reverse proxy

Funkční

Zajišťují více flexibilní integraci, repozitáře, mediátoři (datová/procesová/technická interoperabilita – SOAP), monitorování

Bezpečnostní

Firewally a Gateways

3 Architektura služeb

3.1 Pohledy na službu

- **business** pohled: služba provede efekt *(v reálném světě)*, který přinese reálnou hodnotu uživateli *(např. zakoupení knihy)*
- **konceptuální** pohled: zapouzdření, znovupoužitelnost, loose coupling, abstrakce, composability …
- **logický** pohled: rozhraní služby, její popis a implementace, zaměření na zprávy nebo zdroje
- pohled z hlediska **software architektury**: business service (externí, zveřejněná funkcionalita) x middleware service (interní, technická, zpracovává požadavky)
- pohled z hlediska **technologie architektury**: REST, GraphQL, XML, RPC, SOAP, gRPC, WebSocket, …

3.2 Rozhraní, popis a implementace

Služba může mít více rozhraní, být naimplementována ve více programovacích jazycích.

- **služba**: rozhraní služby + implementace služby
- **WSDL** služba: popis služby v jazyce WSDL
- **SOAP** služba: rozhraní, které je dostupné přes protokol SOAP
- **REST/RESTful** služba: rozhraní, které splňuje styl architektury REST a HTTP protokol
- **mikroslužba**: sada služeb, které realizují schopnosti aplikace

3.3 Public Process

Pracujeme s pomocí stavového diagramu, který popisuje přechod mezi dvěma stavy. To probíhá pomocí operace, která má vstup, podmínky, efekty a výstup.

3.4 Principy služeb

- **loose coupling**: když klient volá službu, měl by na ni být co nejméně závislý
- **reusability**: přepoužitelnost, aby mohla být služba využita v co nejvíce situacích
- **contracting**: domluva jednotlivých stran na rozhraní používaném při komunikaci

- **abstraction**: rozhraní by mělo být technologicky nezávislé a oddělené od konkrétní implementace
- **discoverability**: služba by měla být objevitelná včetně návodu k použití
- **composability**: služby lze složit do více komplexních procesů, které lze znova použít jako služby
- **encapsulation**: zapouzdření, implementace je schována a jde vidět jen rozhraní

3.5 Integrace a interoperabilita

Integrace: spojení aplikací tak, aby mohly sdílet informaci a funkcionality

Interoperabilita: schopnost dvou aplikací, aby se chápaly navzájem na úrovni dat *(syntax/struktura)*, funkcí, procesů a technických aspektů

3.6 SOA

One-To-One integrace: špagetová architektura, vše je propojené neřízeně, je v tom nepořádek.

Many-To-Many integrace: centrální integrace pomocí Enterprise Service Bus, spravuje jednotlivé vrstvy pomocí **SOA**. *(Špagety jsou zde pořád, jen jsou "schované"v ESB)*

SOA je o kultuře, metodologii *(strategie top-down / bottom-up)* a technologii. Jedno z oddělení v IT → ITDelivery řeší **SOA Governance** = propojení jednotlivých systémů.

Integraci provedeme buď přes middleware (M:N, ESB, **service-oriented**), přímo (1:1) nebo přes databázi *(datově orientovaná, aplikace D pracuje s databází aplikace B – pozor na integritu dat).*

Mezi těmito způsoby integrace existuje ještě rozdíl v typu dat: webové služby = **real-time** data // **ETL** *(Extract, Transform, Load)*: načítání dat v dávkách. SOA využívá hlavně real-time, ale také ETL, třeba v případě selhání, synchronizace všech dat.

3.7 ESB

Operační systém → JVM → JDBC, …→ Datové zdroje, JMS → Aplikace. **ESB** je jednou z těchto aplikací, která má sama ještě své podaplikace.

ESB obsahuje služby (sdílené, pro infrastrukturu) i procesy (Technical, Business). **ESB Application** je běžná aplikace na aplikačním serveru. Můžeme zde používat různé integrační vzory.

3.8 Integrační vzory

Synchronní integrace: jeden socket, čas odpovědi na požadavek je krátký, jednoduchý na implementaci, ale zablokuje mi vlákno, server definuje endpoint

Asynchronní integrace: každý požadavek a odpověď má socket zvlášť, jak klient, tak server definují endpoint, čas odpovědi může být dlouhý

Asynchronní komunikace pomocí **prostředníka**: máme frontu požadavků na middlewaru, do které zapisují klienti a ze které čtou servery a frontu odpovědí, do které zapisují servery a ze které čtou klienti. Z tohoto vychází Message Queues a Publish / Subscribe, **používané**.

Asynchronní komunikace pomocí **polling**u: klient otevře socket, server mu potvrdí, že může přijímat a klient ho následně polluje a posílá mu požadavky. → typicky na webu *(server nemůže otevřít požadavek)*.

Message Broker: ESB míchá a spojuje standardní i proprietární způsoby přenosu mezi klienty a servery.

Location transparency: ESB schová změnu v poloze služeb tak, aby např. změna IP adresu serveru nezměnila klienty, dá se použít i pro load balancing.

Session Pooling: ESB udržuje předem daný počet sessionů, které se využívají v runtime, jeden session token může být znovupoužit více instancemi jednoho procesu.

Dynamic Routing: ESB vystavuje službu, které přesměruje na různé služby na základě zpráv.

Message enrichment: Obohatí zprávu předtím, než zavolá nějakou backend službu *(obohacení zprávy o data zákazníka ze služby dat zákazníků)*.

3.9 Datové transformace

Aplikace někdy potřebujeme mezi s sebou namapovat, využíváme strukturu XSLT, XQuery. Při transformaci je třeba namapovat také jednotlivé identifikátory, k čemuž slouží **mapování klíčů** : CRM-ID → UUID → OMS-ID.

3.10 Škálovatelnost

Dělit můžeme 3 způsoby – X (škálujeme napříč instancemi *(uživatel 1-100 sem…)*, Z (datové roz.) a **Y** *(rozdělením na mikroslužby na základě funkcí)*.

Mikroslužby tedy rozdělí aplikaci podle jednotlivých funkcionalit, vytvoří jednotlivé služby, a ty můžou být rozděleny na servery a sdíleny podle potřeby.

Charakteristika mikroslužeb: loose coupling *(klient by neměl vědět o tom, kde se služba nachází)*, technology-agnostic protokol *(nezávisí na konkrétní technologii)*, nezávisle nasaditelné, snadno nahraditelné, zaměřené na informaci *(účetnictví, doporučování)*, implementováno více technologií (polyglot), vlastní ji drobný tým.

4 Aplikační protokoly

Viz. ISO/OSI model nebo TCP/IP model → zajímá nás právě tato aplikační vrstva, která sdružuje aplikační / prezentační / relační vrstvy. Aplikační protokoly závisí především na **TCP** *(ale HTTP/3 už UDP)*.

Většina aplikací je založena na HTTP protokolu, XML-RPC / SOAP jsou založeny na HTTP, WebSocket a Remote Method Invocation (založeno na Javě).

4.1 Socket

Handshaking: ustanovení spojení → server poslouchá na IP:port, three-way handshake (SYN, SYN ACK, ACK), výsledkem je **socket** s unikátními zdrojovými / cílovými adresami a porty.

4.2 Metriky

Vytváření nových spojení je drahé a omezené **latencí** = doba, kdy putují data mezi klientem a serverem *(5 500 km vzdálenost = 28ms)*, proto chceme optimalizovat spojení → HTTP Keep-alive / pipelining.

TCP Fast Open (**TFO**): při prvním navázání spojení se vytvoří TCP Cookie, při dalším se pak pošle toto cookie a spojení se naváže „okamžitě“ *(jen se ověří cookie)* → redukce až o 15%.

Round trip time (**RTT**): doba od poslání prvního požadavku po přijetí (při navazování spojení: 2x latence, poté: *můžou* se tam počítat i čas zpracování požadavku (**SPT**) na serveru) → u nás **RTT** = 2x latence, **RT** *(response time)* = 2x latence + SPT

4.3 Adresace

IP adresa – váže se k rozhraní (eth0, eth1), TCP port je adresou aplikace na jednom rozhraní – více aplikací s různými porty může nalouchat. Existují ale i **aplikační** adresování – například HTTP hlavička Host a domény *(mimo TCP/IP)*.

4.4 HTTP

Aplikační protokol, základ webu; původně jeden socket, request-response, postupně se ale došlo k možnosti využití více socketů, perzistenci, pipelingu, načítání zdrojů (CSS) z více domén *(domain sharding)*.

4.5 HTTP Keep-alive

HTTP keep-alive zajišťuje perzistentní spojení: TCP spojení se použije pro více požadavků a odpovědí, čímž **nemusí** dojít k three-way handshake při každém spojení, dochází k **nižší latenci**. Jednotlivé požadavky se obsluhují ve frontě (FIFO) – *request queue*.

4.6 HTTP pipelining

HTTP pipelining: optimalizace, která umožňuje poslat v jednom síťovém požadavku za sebou **více** HTTP požadavků, které se následně zpracují paralelně *(response queue)*. Pořád zde ale dochází k **head of line blocking** *(dojdou 2 požadavky za sebou, první je ready za 40ms, druhý za 20ms, ale musí čekat, aby došly za sebou)*. Podpora HTTP pipelining je **omezená**, protože mechanismus pro posílání více požadavků za sebou je už implementován v HTTP 2.0.

4.7 Domain sharding

Maximální počet paralelních TCP spojení vůči jednomu originu (protokol; doména; port) pro prohlížeč je **6**. *(tento počet je nastaven kvůli DDoS)*

Tento počet se dá navýšit pomocí tzv. **domain sharding** —> doménu example.com rozdělím na shard1.example.com a shard2.example.com, které ukazují na **stejnou** IP adresu. Na web serveru to rozdělím pomocí konfigurace **VirtualHost**.

4.8 State management

Při **prvním** požadavku na server je v odpovědi hlavička **Set-Cookie**, která nastaví klientovi cookie, kterou následně kopíruje při každém požadavku.

Cookie obsahuje informace o doméně, maximální době platnosti, URL cestu. Klient pak posílá v hlavičce **Cookie**, pokud nevypršela a sedí doména a cesta. Server pak aktualizuje maximální dobu platnosti při každé odpovědi.

Stateful server – server si pamatuje informace o session v neperzistentní paměti serveru (po restartu se smaže).

4.9 SOAP

SOAP je framework pro **posílání zpráv**, založený na XML, o vrstvu výše (nabinduje se na HTTP / SMTP / JMS). Obsahuje odesílatele, příjemce a prostředníky. Obsah SOAP zprávy je obálka s hlavičkou a tělem.

Hlavička obsahuje **metadata** (informace o směrování, rozšíření WS-*). **Tělo** obsahuje vlastní **obsah** zprávy a/nebo **chyby**. Poslední částí je **příloha** pro případ binárních dat.

Endpoint SOAP služby je síťová **adresa** používaná pro komunikaci pomocí požadavků a odpovědí. U **synchronní** komunikace definuje endpoint pouze služba, u **asynchronní** jak služba, tak klient.

4.10 WSDL

Komponenty **WSDL**: informační model *(typy)*, sada operací *(jméno, vstup, výstup, chyby)*, binding *(způsob přesunu zpráv po síti protokolem)*, endpoint *(kde se služba nachází na síti)*. WSDL může být **abstraktní** (pouze informační model a sada operací) nebo **konkrétní**.

5 Architektura aplikačního serveru

Aplikační server: prostředí, kde běží aplikační logika, se kterým komunikuje klient pomocí aplikačního protokolu *(nejčastěji HTTP)*.

Modulární prostředí poskytující technologi pro enterprise systémy, obsahuje různé **komponenty**, kontejnery (JEE, servlety, JMS), poskytuje služby pro výkon, failover *(přepnutí na funkční komponenty)*. Aplikační server **obsahuje** web server v sobě, může být složen z **více** serverů.

Oproti tomu **web server** poskytuje pouze HTTP, zpracovává HTTP požadavky, poskytuje HTTP odpovědi.

5.1 Architektura instance AS

Základem je **operační systém**, na tom běží **JVM**. Nad tím je RMI (volání metod), JDBC (databáze), JMS (zprávy), JNDI (naming & directory interface), JTA (transakce), JMX (e-maily) a další **Java technologie**. Na těch stojí **aplikační server**, který obsahuje datové zdroje, JMS server a další **služby pro aplikace** a nad tím stojí každá **aplikace**.

AS je v operačním systému **jeden proces**, který naslouchá na 1 nebo více IP adresách a TCP portech, jedná se o **Java** proces (funguje zde garbage collection, alokace paměti…)

Pojmy

doména *(skupina serverů se specifickou konfigurací)*, **administrační** server *(spravuje doménu)*, **managed** server *(spouští aplikace a "objekty"[datové zdroje])*, **cluster** *(skupina managed serverů, obsahují stejnou kopii*, **machine** *(fyzický přístroj a OS, na kterém běží servery)*, **node manager** *(proces, který poskytuje přístup k serverům)*, **load balancer** *(síťový prvek, který distribuuje požadavky klientů managed serverům)*.

Servlet

Technologie, která umožní **rozšířit** funkcionality serveru (Java třída, definuje rozhraní, aplikace toto rozhraní implementuje). Jsou používány pro HTTP odpovědi *(HttpServletRequest, HttpServletResponse)*.

5.2 Obsluha požadavku

IO: práce s vstupem nebo výstupem dat, existují dva základní mechanismy implementované různými technologiemi. V tu chvíli, kdy požadavek přijde je vytvořeno nové **vlákno** a řeším dvě úrovně: **inbound** (vstup) a **outbound** (výstup).

Inbound

Synchronní I/O: blokující, pro všechny příchozí požadavky se vytvoří 1 vlákno na 1 požadavek → **nevýhodné**, způsobuje velký počet vláken a nutnost přepínání kontextu

→ **Asynchronní** I/O: předvytvořená vlákna čekají na příchozí požadavky, **obslužná** vlákna jim tyto požadavky přidělují *(jsem schopný to škálovat – jen určitý počet vláken danému požadavku).*

Outbound

→ **Synchronní** I/O: aktivita **pošle** požadavek, a čeká na odpověď, vlákno je uspané *(blocked)* a čeká *(používá se v kombinaci s JVT)*

Asynchronní: I/O: po odeslání požadavku se **nečeká** na odpověď, vlákno může být využito na další požadavky, po získání odpovědi se zavolá **callback**, tzv. event loop.

Komponenty

V rámci aplikačního serveru funguje **muxer**, což je komponenta, která umožňuje zpracovávat komunikaci i přes jiné protokoly (HTTP, RMI…) → předá požadavky do fronty, a z té to potom zpracuje **work manager**.

5.3 JVT

Java Virtual Threads = **virtuální vlákna** *(něco jako korutiny)* – vlákna, která JVM mapuje na vlákna operačního systému, ALE! blokující operace **neblokuje** systémová vlákno.

Úlohy jsou obecně CPU bound, nebo I/O bound (omezeny výkonem CPU nebo I/O), na webovém serveru se jedná o úlohy **I/O bound**.

5.4 RMI

Remote **M**ethod **I**nvocation: protokol, který umožňuje komunikaci mezi **Java** aplikacemi → jedna Javovská třída může vzdáleně **zavolat** metody jiné třídy – používá Java Remote Method Protocol. *(!!! RMI není technologicky nezávislý)*

Fáze **vývojová** *(sdílené rozhraní)* → **kompilace** a spuštění → **vyhledání** referencí na jednotlivé objekty v serverovém registru → následně se **volají** metody na serveru.

Klient = volá vzdálené metody, **server** = poskytuje vzdálené objekty, **stub** = reprezentace komunikačního objektu na klientovi, **skeleton** = reprezentace komunikačního objektu na serveru, **registr** = komponenta držící stuby.

5.5 JNDI

Technologie pro **distribuci objektů** v aplikačním serveru *(hierarchie objektů)*: může držet například strom databází, který se nejprve nakonfiguruje, následně se to přiřadí *(nabinduje)* do struktury, nasadí, a jiný klient je pak schopný tento objekt hledat.

Deployment = klient **nasadí** objekt na **jeden** ze serverů a admin server to následně **rozkopíruje** na všechny další servery v rámci clusteru.

5.6 Object Failover

Aplikační server obsahuje také **failover** mechanismus = v případě výpadku objektu jsem schopný pomocí tohoto mechanismu přepnout na jiný objekt, který **funguje**. *(klient by neměl poznat, že nastal výpadek)*

6 REST

REST není jen protokol, kterým **klient** přistupuje na server, může být využit i pro **crawling**, nebo třeba volání jinou **službou**.

Representational **S**tate **T**ransfer: přenos stavu, obsahujícího reprezentaci, vytvořen Roy Fieldingem v rámci disertační práce, vychází z technických detailů HTTP (bezestavovost).

Služby, které implementují REST v plné podobě, se nazývají **RESTful**, lidé často **porušují** principy RESTu. REST realizuje **WSA** resource-oriented model.

6.1 Principy

Oddělení zodpovědností *(separation of concerns)*: jsem schopný oddělit vrstvy od sebe *(klient, server)*, vývoj může probíhat nezávisle.

Využívání standardů, které definují způsob komunikace, na kterých se shodli uživatelé a organizace.

Open source, bez licenčních poplatků

6.2 Výhody HTTP

Důvěrnost: HTTP je rozšířený, známý

Interoperabilita: HTTP knihovny nalezneme na všech prostředích, můžeme se zaměřit na jádro problému, nezáleží na konkrétní aplikaci.

Škálovatelnost: Webová infrastruktura se dá velmi dobře škálovat (proxy) a cachovat (GET idempotence, safe metody).

6.3 Omezení

REST funguje pouze na architektuře **klient-server**. REST je **bezestavový** *(server by neměl mít uloženou žádnou informaci o session)*, **cachovatelný** = nutnost cachování, **vrstvený systém** – klient by neměl vědět, s kterou vrstvou za endpointem komunikuje, **jednotné rozhraní** (GET, POST, PUT, PATCH, DELETE) – doménově nezávislé.

6.4 Zdroj

Zdroj může být **reálný objekt**, ale také jen abstraktní věc, vzniklá spojením více reálných. Každý zdroj má nějakou svou **reprezentaci** a **identifikátor** tak, aby se k němu klient mohl dostat.

Přístup ke zdroji

Když chce klient přistoupit ke zdroji, musí nejprve provést **dereferenci URI**, aby zjistil, který protokol chce použít. Následně **přistoupí** k tomuto zdroji, služba předzpracuje požadavek a vytvoří **výslednou reprezentaci**, kterou pošle klientovi zpět. Klient tuto reprezentaci následně **interpretuje**.

6.5 Uniform Resource Identifier

URI: identifikuje **zdroj** *(tento zdroj ale nemusí fyzicky existovat)*, může to být URL (lokátor) nebo URN (jméno), je **globálně** platný v rámci internetu. Skládá se ze schématu (není protokol), autority, cesty, dotazu a fragmentu.

→ scheme://authority/path?query#fragment

URL: umožní najít zdroj na zadané **lokaci** v síti, každé URL je zároveň URI

URN: Uniform Resource Name: **jméno**, případně obsahující namespace *(isbn: 9877-1222-1414-1222)*

Resources over Entities

URI často identifikuje zdroj v datovém modelu aplikace: **path** reflektuje datový model (je to graf, URI identifikuje zdroj cestou v tomto grafu).

Oproti tomu **query** umožňuje provést **selekci** *(?status=valid)* nebo **projekci** *(?properties=id,name)*.

Fragmenty jsou definovány na základě formátu = v HTML je to tag **id**, v XML ale nic takového není.

Pojmy

Capability URL: krátkodobá URL pro určitý účel *(odkaz ke změně e-mail adresy)*

URI alias: dvě různá URI identifikující stejný zdroj

URI collision: jedna URI identifikující dva zdroje *(chyba)*

URI opacity: content type jako součást URI

Persistent URL: i po odstranění dokumentu URL zůstává platné

Reprezentace zdrojů

Měly by odpovídat Internet Media Types: XML, HTML, JSON, YAML, RDF

Datový formát: **binární** (specifický, komprimovaná data), **textový** (všechny běžné formáty)

Metadata: data o zdroji, definován HTTP hlavičkami v odpovědi nebo přímo v datovém formátu *(author, updated)*.

Content-Type: Accept (požadavek klienta), Content-Type (odpověď serveru), odpovídá IANA *(Internet Assigned Numbers Authority)* media types, zdroj může poskytnout více reprezentací.

Typické formáty: text/plain, text/html *(data v přirozeném jazyce)*, application/xml, application/json *(specifický formát dané aplikace)* a další specifické zápisy.

Netypické formáty: na začátek podtypu dám **x-**, případně **vnd.**, pokud chci vlastní formát – application/x-latex, application/vnd.ms-excel

Když mluvíme o zdroji, myslíme tím **stav zdroje** (aktuální obsah, který se v průběhu času mění).

6.6 Uniform Interface

Jednotné rozhraní = konečná množina operací, v RESTu se nazývají CRUD (Create – POST/PUT, Read – GET, Update – PUT/PATCH, Delete – DELETE).

Jednotlivé operace **nejsou** doménově specifické: GET nemá sémantiku z pohledu aplikace. *(nejmenuje se to getOrders(), ale GET /orders)*

Vlastnosti method

Safe: nemění stav zdroje, read-only / lookup, lze cachovat (GET)

Unsafe: může změnit stav, transakce, unsafe neznamená nebezpečná.

Idempotence: zavolání metody na stejném zdroji má **stejný** efekt *(dostane zdroj do stejného stavu stavu)* – GET, PUT i **DELETE**

Metody

GET: získá stav zdroje, hledání, cachovatelné, safe, idempotentní, vrátí typicky 200 OK nebo 404 NOT FOUND.

PUT: update (kompletní náhrada) nebo insert (vložení), není safe, ale je idempotentní. Návratový kód 200 OK nebo 204 No Content při aktualizaci, nebo 201 Created při vložení.

PATCH: partial update (částečná náhrada) zdroje, není safe ani idempotentní, návratový kód 200 OK nebo 204 No Content, případně 404 Not Found. Používá se například v **GData** protokolu.

POST: vloží nový zdroj, ID je generováno, klient poskytne pouze obsah a URI, není safe ani idempotentní, 201 Created.

DELETE: smaže specifický zdroj, není safe, ale je **idempotentní** *(vícenásobné zavolání má stejný efekt – zdroj neexistuje)*, návratový kód 200 OK nebo 204 No Content.

HEAD: GET jen pro hlavičky, specifický, safe a idempotentní

OPTIONS: získá konfiguraci zdrojů *(používáno v protokolu CORS)*, safe a idempotentní

Návratové kódy – chyba 4xx

400: obecná chyba na straně klienta, neplatný formát, chyba syntaxe

404: zdroj se zadanou adresou neexistuje

401: nesprávné přihlašovací údaje (user/pass, API klíč), odpověď by měla obsahovat hlavičku indikující typ autenifikace

405: nepovolená metoda HTTP, hlavička Allow umožňuje vypsat podporované metody

406: příliš mnoho omezení na typ přijímaného média (Accept)

Při zpracování bychom **měli** používat návratové kódy! Tedy neexistuje nic jako vrátit 200 s payloadem error. Tak stejně je třeba respektovat sémantiku HTTP metod (tedy nic jako GET /orders/?add).

6.7 HATEOAS

Název vychází z **hypertextu** (reprezentace zdroje obsahující **odkazy**), odkaz je URI zdroje a to, že aplikujeme přístupovou metodu na zdroj pomocí odkazu je **přechod mezi stavy**. HATEOAS umožňuje bezestavovou implementaci služeb.

Stateful server: Stav aplikace je uložen v paměti serveru, klient se identifikuje pomocí cookie.

Perzistentní úložiště: obsahuje data aplikace *(např. databáze zboží)*

Úložiště session: obsahuje stav aplikace, používá se cookies

Stateless server: Nepoužívá paměť serveru, stav se přesouvá pomocí odkazů: POST /orders: obsahuje odkaz /orders/1 → při více serverech je mnohem lepší, protože nemusíme přenášet stav mezi servery.

Odkazy můžou mít hodnotu **rel**, která definuje **sémantiku** operace pod odkazem *(next, previous, self, nebo klidně URI s danou operací)*.

Pokud používám HATEOAS, **nemusím** pak kontrolovat **preconditions** u jednotlivých stavů, protože mi odkazy „nedovolí“ se dostat do neplatného stavu.

Výhody: průhlednost lokace (zveřejníme jen úvodní odkazy, ostatní se můžou změnit beze změny logiky klienta), loose coupling (klient ví, kam se může dostat přes odkazy), bezstavovost (umožní lepší škálovatelnost).

6.8 Caching, Revalidation, Concurrency Control

Škálovatelnost

Na web přichází velké množství požadavků → chceme škálovat, což lze v RESTu pomocí cachování, revalidace a concurrency control.

Caching

Cachujeme vždy **statické** zdroje, ale také **dynamické** zdroje (získávané pomocí GET, pomocí hlavičky **Cache-Control**, podle té se pak revaliduje).

Cache-Control: private *(cachuje pouze klient)*, public *(cachovat může i proxy)*, no-cache *(cachovat se nemá)*, no-store *(nesmí se perzistentně ukládat)*, no-transform *(nesmí se komprimovat data)*, **max-age**: platnost cache (v sekundách).

Last-Modified a **ETag**: umožní cachovat podle posledního data úpravy nebo podle obsahu zdroje (**Strong** ETag = obsah bit po bitu, např. objednávka, **Weak** ETag = sémantický obsah, definuje aplikace, např. u seznamu objednávek, začíná W/) – hlavičky odpovědi.

If-Modified-Since a **If-None-Match** hlavičky požadavku → používá se při revalidaci obsahu *(podmíněný GET – viz. úkol)*.

Concurrency

Používá „optimistické řízení přístupu“ = nezamykám záznam, ale použiji Conditional PUT s hlavičkou **If-Unmodified-Since** *(čas)* a **If-Match** *(ETag)*. Odpovědí pak může být 200 OK nebo **412 Precondition Failed**. Diplomka !!!

Richardson Maturiy Model

Definuje úrovně toho, jak moc je API RESTové:

→ 0. úroveň: pouze **XML** *(používám na vše POST, neexistují zdroje)*
→ 1. úroveň: **zdroje** a URI *(pořád ale ještě používám na vše POST)*
→ 2. úroveň: **HTTP metody** *(používání GET, PUT, DELETE)*
→ 3. úroveň: **odkazy** *(HATEOAS)*

7 Vysoká dostupnost a výkon

7.1 Dobrý výkon

Výkon **je určen** počtem uživatelů, souběžných připojení, zpráv, velikost zpráv, počet služeb, infrastruktura (kapacita, dostupnost, konfigurace).

Vysokého výkonu dosáhneme pomocí **infrastruktury** (škálování, failover), ladění **výkonu** (aplikační server, JVM, OS) a konfigurace služby (paralelní zpracování, optimalizace procesů).

7.2 Definice

Škálování: vytváření víc instancí systému v případě větší zátěže tak, že koncový uživatel nepozná, že systém používá více uživatelů. Může být **horizontální** (přidávám nové servery) nebo **vertikální** (přidávám další CPU, zvětšuji paměť).

Dostupnost: pravděpodobnost, že je služba v daném čase funkční *(99.9987 % = služba má výpadek 44 sekund ročně)*. **LSA**: zaručuje dostupnost služby, jinak zákazník dostane slevu.

Vysoká dostupnost: schopnost systému se škálovat *(když jedna instance spadne, operace budou přesměrovány na jinou)*, **application failover**: když selže jedna komponenta, překopíruje se práce na jinou – musí to být ale možné)

Metrika **doba odpovědi**: kolik času zabere od prvního zadání domény do návratu odpovědi: DNS lookup *(→ nepoužívat doménová jména, ale rovnou DNS)*, TCP handshaking *(→ používat perzistentní připojení)*, RTT *(někdy lze ignorovat)* a samotný server processing time

Další metrika je pak **queries per second** (na straně serveru).

7.3 Load balancing

Rozložení zátěže více instancím aplikace *(na různých strojích, sdílení zátěže)*.

DNS-based load balancer *(DNS round robin)* → NAT-based load balancer → Reverse-proxy load balancer → klientský load balancer.

DNS-based

Na **jeden** DNS záznam je přiřazeno **více** IP adres, DNS systém přiděluje algoritmem Round Robin adresy ze seznamu. Velmi jednoduché, ale není možnost monitorovat status a zdraví serverů.

Reverse Proxy (nginx)

Load Balancer (na úrovni **TCP** a **HTTP**) dostane příchozí požadavek a vybere instanci aplikace, které **přepošle** požadavek. Následně zpětně dostane odpověď a tu **vrátí** zpět. Jednotlivé instance obsahují také **endpointy** pro kontrolu zdraví *(stavu)* aplikace.

Sticky session: load balancer jednomu konkrétnímu uživateli zajišťuje, že bude vždy komunikovat s jedním **konkrétním** serverem *(aby mu nezmizela session → pasivní cookie, nebo aktivní = load balancer přidá vlastní cookie)*.

7.4 Typy Load balancerů

Softwarové: Apache *(mod_proxy_balancer)*, **NGINX**, obsahuje **sticky sessions**, různé možnosti konfigurace, plug-iny nebo hardwarové.

7.5 Round-Robin

Pokud existuje **identifikátor** sessionu, použije se **stejný** server. Jinak se pošle server na **další** server v pořadí *(id serveru, plus modulo)* a zapamatuje si identifikátor sessionu.

7.6 Nastavení proxy

Toto ale nemusí fungovat vždy, server může být přetížen. Proto se používá také metrika **least connections** (požadavek se pošle na server, který je nejméně zatížený) nebo **least time** (požadavek se pošle na server s nejkratší průměrnou dobou odezvy a nejmenším počtem připojení).

Zároveň můžeme omezit **maximální počet** připojení (throttling), nastavit kapacitu, nebo nastavit serveru pomalý začátek *(úmyslné timeouty na začátku, aby server nebyl hned zahlcen)*.

Sticky cookie

Cookie je definováno přímo load balancerem, může ho tedy v případě výpadku přeposlat na jiný server, případně **sticky learn** = load balancer rozpozná cookie od uživatele a využije ho.

Session state persistence

Jinou možností je si ukládat informace o session přímo do databáze, ke které pak přistupují jednotlivé servery, a není tedy třeba používat sticky sessions v load balanceru.

7.7 Monitorování

Potřebujeme sbírat data o běhu systému, filtrovat je, ukládat, zobrazovat a následně **ladit**. Metriky můžeme získat z aplikačního serveru (přístupové logy, server logy), operačního systému (otevřené sockety, paměť, počet přepnutí kontextu, I/O výkon) a databáze.