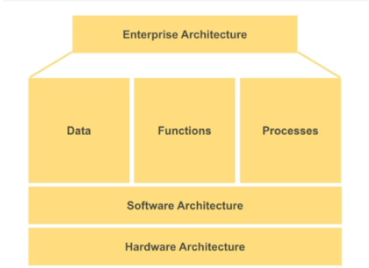
Architektura: popis struktury a chování systému

Z čeho se skládá architektura?
- data *(struktura)* - funkce - procesy *(chování, dynamičnost)*
- software/hardware *(komponenty)*

Funkce a procesy můžeme chápat jako jednu věc, jsou podobné.

Role v systému: zákazník, dodavatel, poskytovatel
Typy architektů:
- technický architekt *(software / hardware úroveň)*
- solution architekt *(navrhuje řešení závislé na konkrétní doméně a technologii)*
- enterprise architekt *(řeší funkčnost systému na úrovních jako BSS)*

Pohled IaaS / PaaS / SaaS



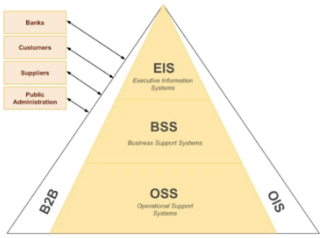
IaaS – infrastruktura *(hardware)*
PaaS – platforma, na které běží systém *(software)*
SaaS – vlastní aplikace *(data, funkce, procesy)*

Složka SaaS je doménově závislá.

Pohled enterprise architektury

EIS *(enterprise information)* – pomáhá vedení organizace
=> BSS *(business support)* – fungují v rámci organizace *(zpracování objednávek, zákazníků)*
OSS *(operational support)* – nejnižší úroveň v organizaci *(kontrola překročení limitu dat)*

OIS *(office information)* – správa dokumentů
B2B *(business to business)* – komunikace
s externími firmami a klienty

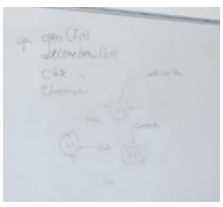


BPEL: jazyk, ve kterém naimplementuji proces
(XML reprezentace volání aktivit, paralelních toků)

Sekvenční diagram volání
- mám jednotlivé swim lanes *(zóny zodpovědnosti – zákazník, AIA – aplikační middleware, BRM [business relationship management] – správa faktur, OMS – správa objednávek)*
- mezi nimi se můžou posílat různé zprávy *(pomocí JMS, SOAP…)*

Data:
- struktura (syntax) – **XML, JSON, YAML**, ORM, datové modely v aplikaci
- význam (sémantika) – záleží na použití *(<user><id/></user> vs <audi><a5/></audi>)*

Integrace: v rámci podniku (aplikace) nebo mezi podniky (B2B)
- integrace na úrovni **dat** vs integrace na úrovni **služeb**
- pro integraci musím použít **rozhraní** *(data – vstup/výstup, funkce – operace, proces – sémantika procesu [stavový diagram], technické detaily [IP, protokol])*

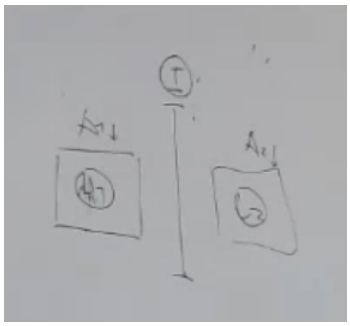


Funkce = otevření objednávky, přidání položky do objednávky, uzavření objednávky.
Proces = postup, kdy musím nejprve otevřít objednávku, pak přidávám položky a uzavřu ji.
Stav = aktuální data (objednávka je otevřena, uzavřena, obsahuje objednávky…)

Typy architektur:
- klient / server, single point of failure *(když server vypadne, je problém)*
  => řešeno mechanismy failover *(při spadnutí)* a škálování *(pro výkon)*
- decentralizované *(Peer-to-peer)*

JDBC: standardní API pro přístup k jakékoliv databázi + ovladače pro komunikaci s databázemi

Separation of concerns:
- definujeme společné rozhraní *(data, procesy)* mezi 2 vrstvami
- umožňuje nezávislou práci oddělených týmů na vrstvách



Klient – server architektury:
- monolitická *(všechny vrstvy na 1 přístroji)*
- dvouvrstvá *(oddělení databáze, škáluji databázi, více uživatelů)*
- třívrstvá *(oddělení aplikačního serveru, tenký prohlížeč)*
- vícevrstvá *(middleware, přidaná hodnota komunikace)*
- mikroslužby *(jednotlivé vrstvy se rozpadnou na více menších, selektivní škálování)*

Typy middleware:
- nefunkcionální // škálovatelnost: load balancery, messaging servery, proxy servery
- funkcionální: procesní servery, mediátoři *(datová, procesní interoperabilita)*
- bezpečnostní: firewall, gateway

Efekt: změna stavu v reálném světě *(dostanu knihu a zaplatím za ni)*

Pohledy na službu:
- business *(jaký má efekt)*
- logický *(resource-view / message-view)*
- technologický *(REST/GraphQL, RMI, RPC, gRPC, Web RPC, SOAP)*
- konceptuální *(loose coupling, encapsulation…)*
- SW architektury *(interní – monitoring, aplikační)*

Služba se skládá z **rozhraní** *(data, funkce, procesy)* a **implementace** *(v daném prog. jazyce)*
+ je rozdíl mezi **popisem** rozhraní a implementací *(JDBC, Java Persistence API, knihovny)*
(ideálně chci jen popsat rozhraní => Swagger, WSDL)

Public Process: stavový diagram, který popisuje volání operace přechodu mezi stavy
*(vstupy a výstupy, podmínky před/po spuštění operace [**preconditions/efekty**])*

Když chci zaplatit objednávku, vstup je zůstatek na účtu, objednávka; podmínkou je, abych měl alespoň takový zůstatek, jaká je cena objednávky; výstupem je, že přijdu o peníze a odešle se mi produkt. Efekty se řeší pomocí podmínek (mohl by být efekt, že nemám záporný zůstatek, ale ten vyřeším tou podmínkou dostatečného zůstatku).

Vlastnosti služeb

Loose coupling: klient by měl vědět o službě co nejméně a být na ní co nejméně závislý
(místo hardcoded adresy v kódu přečtení z konfigurační služby, př. HATEOAS = odkazy na stav, zajištění kompatibility dat se starší verzí – nepovinné/výchozí hodnoty u nových vstupů)

Reusability: přepoužitelnost, služba by měla být využitelná i na dalších projektech *(jako OOP)*

Contracting: domluva stran na rozhraní pro komunikaci *(HTTP Accept hlavička)*

Abstracting: rozhraní je definováno nezávisle na technologii implementace *(HTTP vs ~~RMI~~)*

Discoverability & **composability**: službu bychom měli být schopni „objevovat“ *(zápis popisu služby do registru, tam ji klient objeví a může ji následně zavolat)* a následně na klientovi skládat logiku volání

Integrace: proces propojování aplikací *(v rámci integrace musím zajistit interoperabilitu)*
Interoperabilita: schopnost prvků v rámci procesu, aby si rozuměly
(na jednotlivých úrovních – data, funkce, procesy, technické aspekty)

SOA *(service oriented architecture)*: způsob propojení aplikací v rámci organizace
3 pohledy na SOA:
- kultura *(souvisí s řízením, když něco přepoužijeme, ušetří nám to práci)*
- metodika *(předpis, jak funguje SOA v dané firmě)*
  SOA Governance – popis, jak řídit architekturu SOA
- technologie *(ESB, interoperabilita)*

ESB *(enterprise service bus)*:
- systém, sbírá data z různých systému a umožňuje integraci
- obecný pojem, nejedná se o konkrétní software *(např. SOA suite od Oracle)*

Integrace založená na službách *(v reálném čase, menší množství dat)*:
- přímé propojení služeb (1:1)
- propojení služeb přes middleware (M:N) – *nejprve zavolá transformaci, pak databázi…*

Integrace založená na datech: integruji přímo přes databázi *(služba musí znát strukturu DB)*
ETL *(extract, transform, load)*:
- specializovaný middleware, zpracování dat
- použití: při výpadku systému, sladění *(reconciliation)* – synchronizaci služeb

Asynchronní komunikace:
- skrze prostředníka *(fronta zpráv – JMS, publish / subscribe)*
- přes polling *(v intervalech se kontroluje stav zpracování požadavku, na webu)*

Integrační vzory middleware

Message broker: ESB dokáže komunikovat s klienty i servery pomocí více technologií *(SOAP, HTTP, JMS, FTP, REST, soubor, proprietární technologie)*

Location transparency: zaručuje loose coupling, klient přistupuje ke službě přes ESB, při změně adresy služby to neovlivní klienta

Session pooling: ESB předvytvoří sadu spojení *(session)* do databáze / na službu, které pak znovuvyužívá *(při častých přístupech se tak nemusí znovu navazovat spojení)*

Dynamic routing: ESB na základě podmínek přeposílá požadavek na službu *(v DÚ)*

Message enrichment: ESB obohatí zprávu o informace z jiné služby před voláním další služby

Datová transformace: návrh *(namapování schémat, XSLT)* + samotné provedení

Key Mapping:
- každá integrovaná aplikace má svou databázi, ve které má speciální klíč
- middleware vytvoří mapovací tabulku s univerzálním ID a identifikátory aplikací
- pokud někde budou chybět identifikátory, měli bychom provést rekonciliaci

The Scale Cube: při škálování aplikace můžeme škálovat třemi způsoby
- osa X: škálujeme z pohledu výkonu *(vytváříme nové instance aplikace + load balancer)*
- osa Y: škálujeme rozdělením na mikroslužby *(rozdělíme skupinu funkcí na menší funkce)*
- osa Z: škálujeme z pohledu rozdělení na základě dat *(data o uživatelích ve více databázích)*

Výhody architektury mikroslužeb: loose coupling,
- protokoly pro komunikaci by měly být nezávislé na technologii *(HTTP)*
- nezávisle instalovatelné, jednoduše nahraditelné, rozděleny podle funkcí

Aplikační protokoly: HTTP, RMI, WebSocket *(aplikační)*, TCP *(transportní)*, IP *(síťová vrstva)*

Komunikace:
- TCP handshake (SYN, SYN ACK, ACK) – výsledkem je socket [sip,sport:dip,dport]
- optimalizace: TCP Fast-Open *(umožní rychlejší navázání spojení, na úrovni kernelu)*, HTTP Keep-Alive *(nemusí se znovu dělat handshake)*, HTTP pipelining

Latence: doba, než data z klienta dojdou na server *(nebo ze serveru na klienta)*
RTT *(round trip time):* doba, než data dojdou z klienta na server a zpět (2x latence)
SPT *(server processing time)*: doba, kterou bude server zpracovávat požadavek
RT *(response time)*: RTT + SPT = doba od odeslání požadavku po přijetí odpovědi

Perzistentní spojení (HTTP Keep-Alive): neděláme znovu TCP handshake, využijeme existující socket => ušetříme na jeden požadavek jeden RTT

HTTP pipelining: požadavky se zpracovávají paralelně ve frontě za sebou, ale může dojít k:
Head of line blocking: stav, kdy musíme čekat na zpracování požadavku, který přišel dřív *(požadavek HTML, pak CSS; CSS je hotové dříve, ale odpověď se může poslat až po HTML)*

Vůči jednomu originu *(protokol, doména, port)* může být maximálně **6** paralelních spojení *(ochrana před DoS útoky, výchozí nastavení)*

Domain sharding: dvě různé domény směřují na stejnou IP adresu, umožní navýšit počet paralelních spojení

Virtual Host: nastavení Apache, které umožní přistoupit k jedné stránce přes 2 různé domény

```
<VirtualHost *:80>
        ServerName www.example.com
        ServerAlias shard1.example.com shard2.example.com
        ServerAdmin admin@example.com
        DocumentRoot /var/www/apache/example.com
</VirtualHost>
```

Testování protokolu HTTP: curl, wget, telnet

State management: header Set-Cookie, Cookie *(doba platnosti Max-Age, doména, URL Path)*
Stateful server: pamatuje si informace o session *(v neperzistentní paměti)*

SOAP: leží nad aplikačními protokoly *(ale narozdíl od RESTu nezávisí konkrétně na HTTP, nelze tedy použít např. Cookie)*

SOAP závisí na **XML**, nepodporuje jiné datové formáty

SOAP zpráva:
- envelope: obsahuje zprávu samotnou
- header: obsahuje metadata a hlavičky *(WS-*)*
- body: data zprávy ve formátu XML *(i chyby)*
- attachment: ne-XML přílohy *(binární data)*

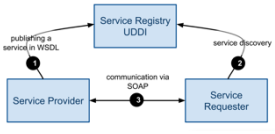
```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header />
    <env:Body>
    <hello xmlns="http://services/" />
  </env:Body>
</env:Envelope>
```

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:helloResponse xmlns:ns2="http://services/">
      <return>hello</return>
    </ns2:helloResponse>
  </S:Body>
</S:Envelope>
```

SOAP endpoint: adresa používaná pro komunikaci *(neobsahuje žádná data, zdroje, informace)*

WSDL: dokument popisující rozhraní služby

Synchronní komunikace: endpoint definuje server
Asynchronní komunikace: endpoint definuje i klient *(může být problém v případě privátních sítí, využívá se v hlavičce WS-Addressing, ReplyTo)*

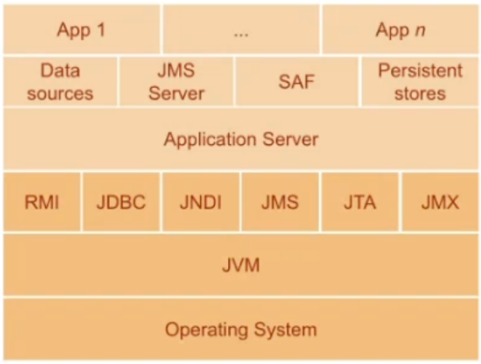


Obsah WSDL dokumentu: datový model *(types)*, funkce a procesy *(portType)*, způsob přenosu konkrétními protokoly *(binding)*, endpoint *(service)*

```
<types>
    <xs:schema xmlns:tns="http://services/" xmlns:xs="http://www.w3.org/2001/XMLSchema"
        <xs:element name="hello" type="tns:hello"/>
        <xs:element name="helloResponse" type="tns:helloResponse"/>
        <xs:complexType name="hello">
            <xs:sequence/>
        </xs:complexType>
        <xs:complexType name="helloResponse">
            <xs:sequence>
                <xs:element name="return" type="xs:string" minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:schema>
</types>
<message name="hello">
    <part name="parameters" element="tns:hello"/>
</message>
<message name="helloResponse">
    <part name="parameters" element="tns:helloResponse"/>
</message>
<portType name="TestService">
    <operation name="hello">
        <input message="tns:hello"/>
    <output message="tns:helloResponse"/>
    </operation>
</portType>
<binding name="TestServicePortBinding" type="tns:TestService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="hello">
        <soap:operation soapAction=""/>
        <input>
            <soap:body use="literal"/>
        </input>
        <output>
            <soap:body use="literal"/>
        </output>
    </operation>
</binding>
<service name="TestServiceService">
    <port name="TestServicePort" binding="tns:TestServicePortBinding">
        <soap:address location="http://127.0.0.1:8080/WS/TestServiceService"/>
    </port>
</service>
```

Aplikační server: prostředí, které lze použít pro běh aplikací s pomocí aplikačních protokolů
Webový server: aplikační server, pouze HTTP

Architektura aplikačního serveru: viz obrázek



Základní technologie aplikačního serveru:
- RMI *(vzdálené volání metod, zveřejnění rozhraní [stub], přenos na klienta, komunikace)*
- JDBC *(rozhraní pro přístup do databáze)*
- JNDI *(distribuce objektů)*
- JMS *(technologie pro posílání zpráv)*
- JTA *(konfigurace transakcí přes datové zdroje)*
- JMX *(API pro správa objektů)*

Doména: skupina serverů se specifickou konfigurací
Administrační server: server, který spravuje doménu
Managed server: server, který spouští aplikace a objekty *(např. datové zdroje)*
Cluster: skupina managed serverů, obsahují stejnou kopii
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 interface)*

Synchronní inbound I/O: blokující, jeden příchozí požadavek = jedno vlákno
Asynchronní inbound I/O: předvytvořená vlákna čekají na příchozí požadavky, obslužná vlákna jim tyto požadavky přidělují *(jsem schopný škálovat, n vláken danému požadavku)*

Synchronní outbound I/O: aktivita pošle požadavek, čeká na odpověď v *blocked* stavu, používá se v kombinaci s JVT *(Java Virtual Threads)*
Asynchronní outbound I/O: po odeslání požadavku se nečeká, po získání odpovědi se zavolá callback

Muxer: komponenta, která zpracovává komunikaci přes různé protokoly *(HTTP, RMI)*
JVT: virtuální vlákna, JVM je namapuje na systémová *(uspání neblokuje systémové vlákno)*

RMI: protokol pro komunikaci mezi Java aplikacemi, není technologicky nezávislý
- server nahraje do registru **stub** *(reprezentaci pro klienta)*, u sebe má **skeleton**
- klient, který chce komunikovat, si stáhne z registru daný stub
- při komunikaci se stub přenáší do podoby pro přenos po síti – marshalling / unmarhsalling

JNDI: drží strom objektů, které se nasazují na jednotlivé managed servery

Object Failover: mechanismus **replicate-aware stub**, kdy je v případě výpadku objektu přepnuto na jiný, fungující objekt *(například ve stejném clusteru)*, klient tedy nepozná, že došlo k výpadku

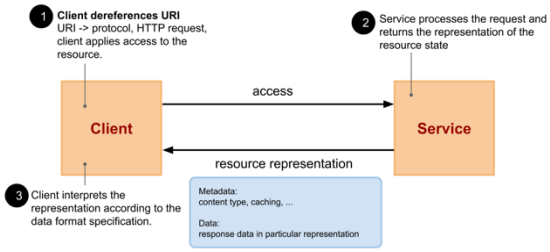
REST: typ architektury, založený na přenosu stavu **zdroje** obsahujícího reprezentaci
RESTful služba: služba implementující REST v plné podobě

Principy RESTu:
- separation of concerns: rozdělení klienta a serveru umožňující nezávislý vývoj
- využití standardů, na kterých se shodli uživatelů a organizace
- open source

Výhody RESTu:
- založen na rozšířeném protokolu HTTP
- **interoperabilita**: knihovny pro práci s HTTP nalezneme ve všech prostředích
- **škálovatelnost**: webová infrastruktura se dobře škáluje *(proxy, cache)*

Omezení (constraints) RESTu:
- funguje pouze na architektuře klient – server
- bezestavovost *(server by neměl mít uloženou žádnou informaci o session)*
- cachovatelnost, vrstvený systém, jednotné rozhraní (uniform interface)

Zdroj:
- jakýkoliv reálný objekt nebo abstraktní věc vzniklá spojením více reálných
- má svou reprezentaci a identifikátor
- přistupujeme k němu **dereferencí URI**



URI: identifikátor zdroje, URL nebo URN
URL: lokátor, kde na síti se daný zdroj nachází
URN: jméno zdroje, globálně platné v rámci internetu

```
URI = scheme ":" [ "//" authority ] [ "/" path ] [ "?" query ] [ "#" frag ]
```

Schéma: nemusí být obecně protokol, může to být i **isbn**: nebo třeba **mailto**:
Autorita: doména nebo adresa serveru
Path: hierarchická podoba /recept/1
Query: klíč=hodnota, umožňuje **selekci** (?status=valid) nebo **projekci** (?properties=id,name)
Fragment: reference na sekundární zdroj v rámci primárního zdroje, závisí na konkrétním formátu *(prvek s daným id na HTML stránce, v XML není definována sémantika)*

Capability URL: krátkodobá URL pro určitý účel *(odkaz ke změně e-mailové adresy)*
URI alias: dvě různá URI identifikující stejný zdroj
URI collision: dva různé zdroje identifikovány jednou URI *(chyba, nemělo by nastat)*
URI opacity: datový formát *(content type)* je součástí URI *(http://example.org/customers.xml)*
Verze zdroje kódovaná v URI: *http://example.org/v1/customers*
Persistent URL: URL adresa, která i po odstranění dokumentu zůstane platná *(a přesměruje)*

Reprezentace zdrojů:
- měla by odpovídat Internet Media Types: XML, HTML, JSON, YAML, RDF
- binární nebo textový formát
- samotná data, metadata *(HTTP hlavičky v odpovědi, přímo v zdroji)*, reprezentace

Content negotiation: domluva klienta a serveru na použitém datovém formátu, umožněno HTTP hlavičkami Accept *(požadavek klienta)* a Content-Type *(odpověď serveru)*

Typické formáty:
- text/plain, text/html *(data v přirozeném jazyce)*
- application/xml, application/json *(specifický formát dané aplikace)*
- application/x-latex *(specifický formát, který není definován v IANA)*
- application/vnd.ms-excel *(formát specifický pro daného poskytovatele)*

Stav zdroje: aktuální reprezentace stavu zdroje

Uniform interface: konečná množina doménově nezávislých operací
*(**C**reate – POST/PUT, **R**ead – GET, **U**pdate – PUT/PATCH, **D**elete – DELETE + HEAD/OPTIONS)*

Vlastnosti metod:
- safe: nemění stav zdroje, lze cachovat *(GET, HEAD, OPTIONS)*
- unsafe: může změnit stav zdroje, neznamená to, že je metoda nebezpečná
- idempotent: vícenásobné volání metody má stejný **efekt** *(dostane zdroj do stejného stavu => GET, PUT, DELETE)*

Je dobré znát vlastnosti a typickém použití jednotlivých metod a návratových kódů.

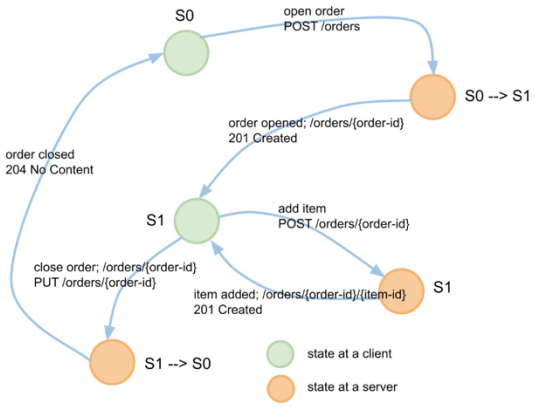
Časté prohřešky:
- nerespektování sémantiky HTTP metod *(GET /orders/add, 200 OK {"msg":"unauthorized"})*
- může vést k problémům *(proxy servery cachují GET požadavky = dvojité přidání)*

Richardson Maturity Model: definuje úrovně toho, jak moc je API RESTové
- 0. úroveň: pouze XML, neexistují zdroje *(POST /getCustomers, /createOrder)*
- 1. úroveň: zdroje a URI *(pořád ale používám na vše POST)*
- 2. úroveň: HTTP metody
- 3. úroveň: HATEOAS

HATEOAS: reprezentace zdroje obsahující odkazy (hypertext) umožňující přechod mezi stavy
- způsob bezestavové implementace služeb *(x stateful server)*

Perzistentní úložiště: obsahuje data aplikace
Session memory: obsahuje stav aplikace, používá cookies

Stateless server: nepoužívá session memory, stav se přesouvá pomocí odkazů




```
<order a:xmlns="http://www.w3.org/2005/Atom" xmlns='
    <id>2324</id>
    <a:link rel="http://company.com/op/addItem"
        href="http://company.com/orders/2324"/>
    <a:link rel="http://company.com/op/deleteOrder"
        href="http://company.com/orders/2324"/>
</order>
```

Reprezentace:
- Atom formát (XML) – standardně **rel** next, previous, self nebo rozšířené
- odkazy definované v http hlavičkách (XML, JSON)
- odkazy v JSON formátu

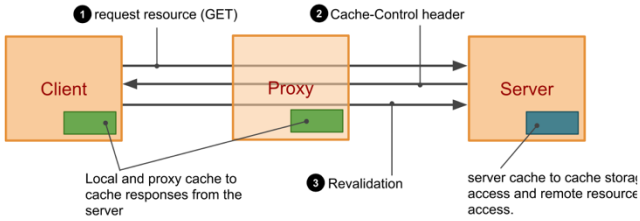
```
> HEAD /orders HTTP/1.1

< Content-Type: application/xml
< Link: <http://company.com/orders/?page=2&size=10>; rel="next"
< Link: <http://company.com/orders/?page=10&size=10>; rel="last"
```

Výhody HATEOAS:
- nemusím řešit sticky sessions při použití load balancery
- nemusím kontrolovat preconditions *(zobrazím pouze ty odkazy, které jsou validní)*
- loose coupling *(klient nemusí znát předem význam jednotlivých stavů)*
- location transparency *(odkazy reprezentují pohled aktuálního uživatele včetně práv)*

Škálovatelnost: na web přichází velké množství požadavků, chceme škálovat
- caching, concurrency control

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



Conditional GET: umožní cachovat podle posledního data úpravy nebo obsahu zdroje
- **If-Modified-Since**: pokud zdroj nebyl upraven po zadaném datu, vrátíme 304 Not Modified
- **Weak ETag**: pokud zdroj nebyl upraven *(obsah určen sémanticky, aplikací)*, začíná W/
- **Strong ETag**: pokud zdroj nebyl upraven *(obsah bit po bitu)*, hlavička **If-None-Match**
- server by měl v odpovědi vždy vrátit i **Last-Modified** a **ETag** *(pro kontrolu)*

(typicky složené zdroje, např. seznamy používají weak ETag, jednoduché strong ETag)

Conditional PUT: aktualizuje zdroj pouze pokud nebyl změněn *(optimistické řízení přístupu)*
- **If-Unmodified-Since** (nebyl změněn na základě času), **If-Match** (na základě **strong** ETagu)
- v případě úpravy vrátí server 412 Precondition Failed

Výkon: ovlivněn počtem uživatelů, souběžných připojení, zpráv, služeb, infrastrukturou

Prostředky k dosažení lepšího výkonu:
- infrastruktura *(škálování, failover)*
- ladění výkonu *(aplikační server, JVM, OS)*
- konfigurace služby *(optimalizace procesů, paralelní zpracování)*

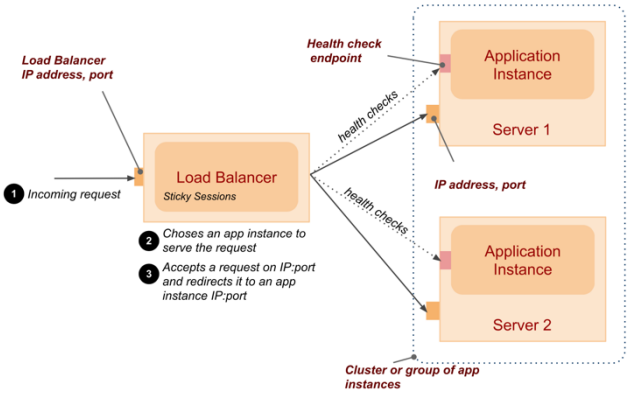
Škálování: vytváření víc instancí systému v případě větší zátěže tak, aby koncový uživatel nepoznal, že je systém zatížen

Vertikální škálování: přidávám další CPU, zvětšuji paměť
Horizontální škálování: přidávám nové servery

Dostupnost: pravděpodobnost, že bude služba v daný čas dostupná
SLA: zaručená dostupnost dané služby, pokud je dostupnost nižší, zákazník dostane slevu

Application failover: když selže jedna komponenta, překopíruje se práce na jinou
Load balancing: rozložení zátěže mezi více serverů a mezi více instancí aplikace/objektu

DNS based:
- na jeden DNS záznam je přiřazeno více IP adres
- DNS systém přiděluje Round Robinem adresy *(pokud už klient komunikuje, přepošle na stejný server, jinak na další v pořadí)*
- není možnost monitorovat status a zdraví serverů



Reverse proxy:
- load balancer si udržuje informace o zdraví *(stavu)* jednotlivých endpointů
- příchozí požadavek je přeposlán na jednu z běžících instanci

Sticky session: load balancer jednomu konkrétnímu uživateli zajistí, že bude vždy komunikovat s jedním konkrétním serverem *(aby mu nezmizela session)*

- pasivní cookie: použije se session ze serveru, load balancer ji pouze „pozoruje“
- aktivní cookie: load balancer přidá vlastní cookie

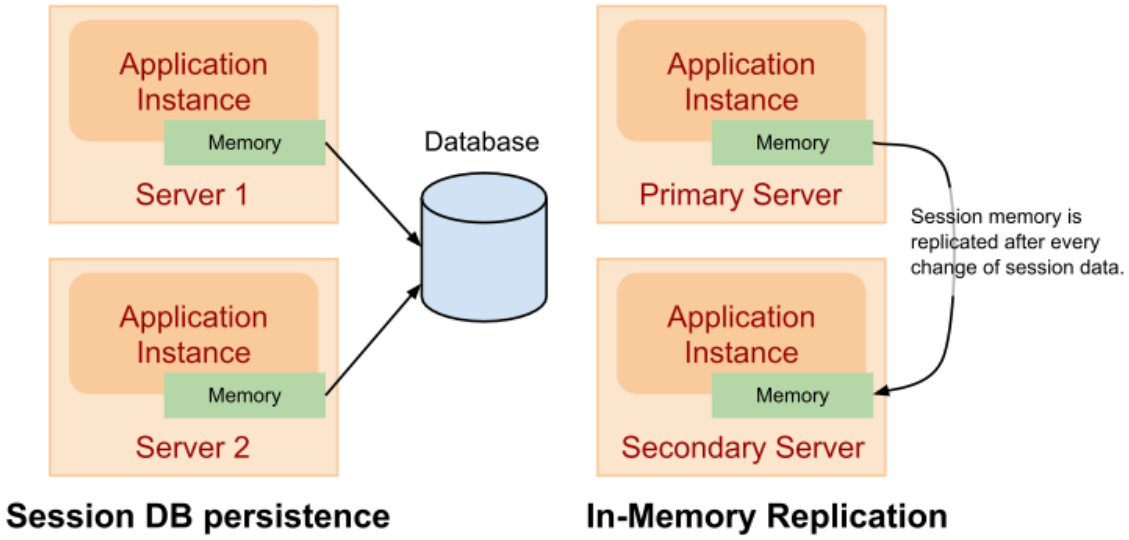
Nastavení proxy:
- least connections: požadavek se pošle na server, který je nejméně zatížený
- least time: požadavek se pošle na server s nejkratší průměrnou dobou odezvy a nejmenším počtem připojení

```
http {
    upstream backend {
        server backend1.example.com weight=5;
        server backend2.example.com;
        server 192.0.0.1 backup;
    }

    server {
        location / {
            proxy_pass http://backend;
        }
    }
}
```

Throttling: omezení maximálního počtu připojení na jeden konkrétní server
Server Slow-Start: úmyslné time outy serveru na začátku, aby nebyl při startování zahlcen

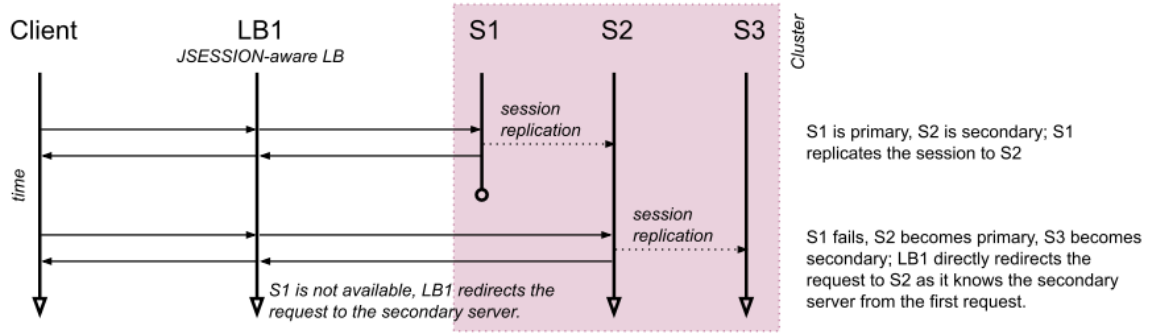
Monitorování: sbírání dat o běhu systému, filtrování, ukládání a ladění
- data o běhu jsou dostupná v přístupovém a server logu aplikačního serveru a databáze
- statistiky operačního systému *(otevřené sockety, paměť, počet přepnutí kontextu)*



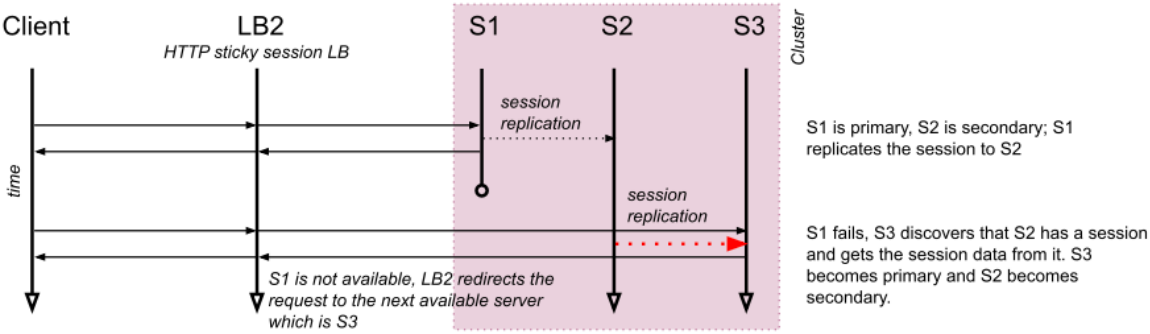
Perzistence stavu session:
- stav se ukládá v databázi, není potřeba používat sticky sessions v load balanceru

In-memory replication:
- primární server drží stav session, sekundární server drží její repliku
- `JSESSIONID=SESSION_ID!PRIM_SERVER_ID!SEC_SERVER_ID!CREATION_TIME`
- pokud primární server běží, load balancer zde přepošle požadavek, jinak na sekundární

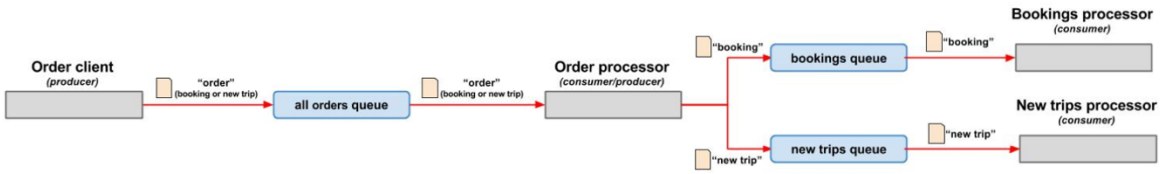
Scenario A: JSession-aware load balancer



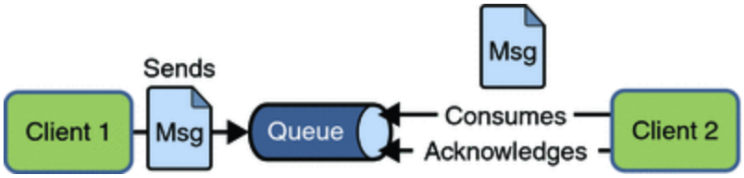
Scenario B: HTTP sticky session load balancer



Bonus: JMS fronty *(obrázek)*



Point-to-Point



Publish/Subscribe