Principy Web 2.0:
- read/write web: uživatel není pouze konzument
- programmable web: myslíme nejen na prohlížeč, ale také na API a možnosti rozšíření
- realtime web: využití informací v reálném čase
- social web: sociální sítě, sdílení dat *(ad DDW)*

Preemptivní multitasking: operační systém určí, jak dlouho poběží které vlákno a kdy dojde k přepnutí kontextu, použito u blokujících úloh
Kooperativní multitasking: pokud běžící úloha není blokující, ale dělá jen jednoduché operace
- úloha se může pozastavit, když čeká, během toho můžou běžet jiné úlohy

Javascript Runtime: každý tab v prohlížeči má svůj runtime
- fronta *(zprávy k zpracování – data, callback, zpracování jedna po druhé)*`
- zásobník
- halda *(paměť)*

Pořád platí, že lze mít současně max. 6 spojení vůči jednomu originu *(viz sharding)*
Všechny požadavky k vykonání se posílají do **page request queue**

JS Worker: má vlastní event loop, dedikovaný / sdílený, k vykonávání jakýchkoliv **blokujících** činností
Callback hell: volání hodně callbacků v sobě, řešení: promise / async await
Promise: stav pending / fullfilled / rejected, slouží k asynchronním voláním

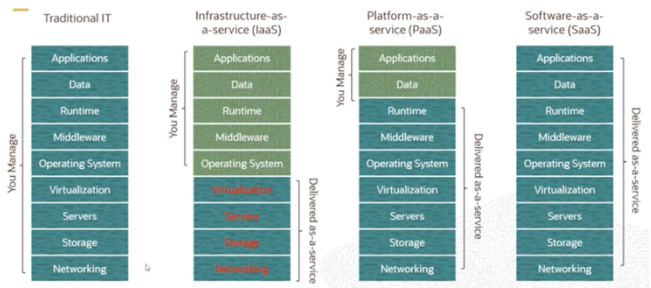
DevOps: umožní automatizaci procesu vývoje / testování / nasazení
Cloud Native: mikroslužby, provozované jako kontejnery

Aspekty cloudu: dostupnost, spolehlivost, nízká cena *(pay per use)*, elasticita (škálovatelnost)
Cloud: jiný způsob přemýšlení, vyžaduje důvěru v poskytovatele, musí umožňovat:
- škálovatelnost výkonu / úložiště, programové rozhraní (API)

CAPEX / OPEX: capital / operational expenditure: u klasického on-premise řešení platíme pevnou cenu za HW, musíme platit za údržbu vs v cloudu platíme podle skutečného využití, údržbu řeší provozovatel cloudu

On-demand: uživatel si sám požádá o zdroje, když je potřebuje (**self-service**)
Broad network access: přístup přes síť
Pay per use: platíme pouze za to, co skutečně využijeme
Lift & shift: vezmu aplikaci z on-premise serveru a 1:1 ji zreplikuji na cloud
Škálovatelnost: možnost automatizovaně přidávat zdroje a nafukovat infrastrukturu podle potřeby

Modely služeb (Cloud Layers):
- IaaS: infrastruktura, zdroje *(HW)*
- PaaS: jen platforma *(Kubernetes)*
- SaaS: jen software *(aplikace)*



Modely nasazení:
- public cloud *(Google, Amazon)*
- private cloud *(vlastní prostředky, on-premise – OpenNebula, VMWare)*
- hybridní cloud *(část public/private)*

IaaS: load balancer, auto-škálování, monitoring / PaaS: Kubernetes / SaaS: Google Apps

Multitenancy: architektura, kde jsou zdroje (vše nebo infrastruktura – síť) sdílené mezi více uživatelů
=> vede k úspoře zdrojů, ale obtížná situace ve špičce

Sdílení pomocí virtuálních zdrojů: sdílená infrastruktura *(VMs)*, oddělení zajišťuje hypervizor
Sdílení pomocí virtualizace operačního systému: sdílená infrastruktura, spravuje operační systém
- mezi aplikací a operačním systémem není abstrakce
Sdílení všech zdrojů: SaaS, všichni 1 zdroj, různí uživatelé jsou od sebe izolováni na úrovni aplikace

Infrastruktura: prostředí pro běh aplikace

Tenancy: prostor pro uživatele (nájemníky, tenants) v cloudu, v něm vytváříme naši infrastruktur
Datové centrum: budovy s racky propojenými fyzickou sítí, napájení včetně záložního zdroje

Potřebujeme: servery *(umístění)*, konektivitu, úložiště, identity management *(cloudu vs aplikace)*
- monitorování, logování, audity *(cena)*

Virtual Cloud Network (VCN): virtuální síť, v rámci této sítě vytvářím jednotlivé **podsítě**
- potřebuji CIDR *(classless inter-domain routing)* adresu
- pro přístup do soukromé sítě se použije DNAT, přístup ven = SNAT, mezi podsítěmi = route table

Podsítě:
- veřejná: bez omezení *(nebezpečné)*
- **soukromé**: kompletně izolované nebo komunikace pouze do internetu (ven)

Peering: propojení dvou VCN – lokální (v rámci 1 regionu) / remote (mezi více regiony, pomalejší)

Shape: množství paměti a CPU, které dostanu na cloudu přiděleno

Virtuální stroje: multi-tenantní model *(hypervizor)* vs **bare meta**l: single tenant *(přímý přístup k HW)*
Dedikované virtuální stroje: single-tenant, nesdílené

Load balancer: kontroluje zdraví jednotlivých backendů, zaručuje vysokou dostupnost

Image: šablona pro virtuální disk s OS, uložen na boot volume, výchozí / custom image

Automatické škálování: upraví počet výpoč. instancí podle zátěže, cooldown pro škálování up/down

Lokální NVMe SSD: lokálně připojené, bez RAIDů, problematické zálohy a snapshoty
Blokové disky: NVMe SSD, diskové servery, více replik na více serverech
Souborové úložiště: NFS *(síťový souborový systém)*, klient se připojí pomocí **mount**
Objektové úložiště: ukládá nestrukturovaná data *(velké velikosti – logy, image disku)*

Jmenný prostor: logická entita pro buckety a objekty
Bucket: samotný kontejner /n/*namespace1*/b/*bucketx*/o/*img1*
- hot bucket = přístupný hned
- **cold bucket**: přístup k datům jako jsou logy, data musím ukládat min. 90 dní
  doba přístupu u cold bucket (**Time To First Byte**) je až 4 hodiny

IaaC: infrastructure as a code – způsob, jak by se měly infrastruktury vytvářet
- kdysi: přístup přes uživatelské rozhraní, IaaC: přístup pomocí konfiguračních souborů (**skriptů**)

Konfigurační nástroje: infrastrukturu mám, chci si do ní nainstalovat balíčky (Ansible, Chef, Puppet)

Cloud Native: proces přesunu legacy aplikací do cloudu, nutný re-design a refactoring aplikace
- výhody: loose coupling, snadnější provádění častých změn

Trail Map: proces přesunu na Cloud Native
1. kontejnerizace *(kontejner != mikroslužba)*
2. CI/CD *(zavedení automatizovaných procesů)*
3. orchestrace
4. pozorování, analytika *(monitorování, logy)*

Lift and Shift: vezmu starou aplikaci, rovnou ji dám do cloudu, jednodušší škálování než on-premise
- výhody: méně úvodních nákladů *(CAPEX / OPEX)*

Mikroslužby: na aplikačním serveru máme různé služby/aplikace
Rozdíl mikroslužeb oproti architektuře monolitu:
- loose coupling, nezávislé na technologií, nezávisle nasazovatelné, různé technologie

Kontejnery: základem může být hardware / virtuální stroj, izolované, sdílí OS, binárky a knihovny, API
Docker: používá Linux NS, unified filesystem (umožňuje vytvářet, sdílet obrazy systému), vícevrstvý

Docker – Container engine: přijímá vstup (CLI / API), stahuje image, připravuje metadata
Docker – Container runtime: abstrakce systémových volání, komunikuje s kernelem OS

Image: souborový systém pro kontejner, v průběhu běhu systému je neměnný *(immutable)*
Kontejner: proces(y) sdílící Linux NS

Klient: aplikace, která komunikuje přes API
Registr: služba pro ukládání image *(Docker Hub)*
Swarm: cluster více docker enginů

Linux namespaces: izolace Linux procesů *(nevidí se vzájemně)*, po spuštění každý proces ve výchozím

net: v síťovém namespace se vytvoří celý stack s NAT
- více služeb tak může fungovat na jednom portu, tento port se pak může namapovat ven (EXPOSE)
mnt: izoluje mount pointy souborového systému => kontejnery si nevidí do svých souborů
uts: hostname, **ipc**: fronty, semafor, paměť, **pid**: izoluje procesy v kontejnerech *(každý má 1)*

user: izoluje UID uživatelů a GID skupin => zabrání elevaci oprávnění a neoprávněnému přístupu
cgroup: umožní izolovat systémové prostředky

Obsah image: kernel, image, příkazy (ADD), kontejner
OverlayFS: umožní mount jednotlivých filesystémů
Image layering: umožní načíst image z registry a vrstvit je

Práce s Dockerem: vytvoří se image, spustí se na něm kontejner, v kontejneru se pouští příkazy, zastaví se, smažeme kontejner, smažeme image

Dockerfile: skript pro vytvoření nového obrazu, můžeme nahrávat obecně do registry

Síť, propojování: bridge *(připojí se do sítě hosta přes podsíť)*, host *(hostovská síť)*, none *(vlastní síť)*

Data Volume: perzistence dat kontejneru, sdílení mezi kontejnery
- adresáře z hostitele namapované do kontejneru, pozor na elevaci oprávnění *(přístup jako root)*

Kubernetes: platforma umožňující provoz kontejnerizované aplikace na clusteru
Vlastnosti:
- automatic binpacking: systém kontejnery pustí automaticky na dostupných uzlech *(CPU, paměť)*
- horizontální škálování, automatic rollout / rollback, health monitoring
- orchestrace úložiště *(automatická správa filesystemu)*, self-healing *(auto restart v případě výpadku)*
- service discovery, load balancing *(rozdělení IP adres mezi jednotlivé instance)*

Architektura: control plane *(řídící uzly)* + výpočetní uzly, manager, etcd (*úložiště*), proxy, scheduler

Control Plane scheduling: pouští komponenty na nodech, detekce a odpověď na události na clusteru
Pod: množina kontejnerů, sdílející stejné namespaces
Kubernetes namespace: množina logicky organizovaných zdrojů v rámci aplikace

kube-apiserver: vystavuje Kubernetes API, **etcd**: key-value úložiště pro cluster data
kube-scheduler: přiřazuje uzly, na nichž poběží nově vytvořené Pody *(úložiště, omezení, specifikace)*
kube-controller-manager: řeší výpadek uzlů, endpointy, vytváří Pody pro joby *(one-time task)*
cloud-controller-manager: interaguje s cloud zdroji, nastavuje VLAN *(uzly/route, load balancery)*

Kubelet: agent pro správu uzlu v clusteru
Kube-proxy: spravuje konfiguraci sítě
Runtime: spouští kontejnery

Pod (zdroj): skupina 1/více úzce svázaných kontejnerů, sdílí úložiště, síť, jedna instance aplikace
- **typicky** v 1 podu běží 1 jeden kontejner
Service (zdroj): řeší networking a zdroje, každý pod je součástí některé service

Workload (zdroj): vytváří Pody podle konfigurací
- deployment *(stateless aplikace)* / stateful set *(zajistí stavovost)*
- job *(skript, př. vytvoří DB schéma)*, cron job, daemon set *(pustí aplikaci na každém node)*

Minikube: umožní testování, demo na 1 přístroji
Kubectl: CLI přístup ke Kubernetes clusteru

Prohlížeč: mnoho komponent (HTML parsing, DOM), univerzální platforma pro doručování webových aplikací, **networking stack**: HTTP/1.1, HTTP 2, caching, same origin policy *(pod tím TCP, TLS, UDP)*

Socket management: v rámci prohlížeče – socket manager (drží si socket pool podle originů (6/org)
- požadavky se zařazují do request queue a přiřadí se jednotlivým socketům

Network Security: aplikace nemají přístup k socketům *(port scanning)*
- TLS negotiation: kontrola platnosti certifikátů (LetsEncrypt)
- same-origin policy: omezení požadavků mezi originy

Mashups: aplikace využívající více zdrojů z různých originu *(XHR / fetch API)*
- **data**: pouze čtou, **service**: čtou i zapisují, **vizualizační**: přečtu data, zobrazím je na mapě

XMLHttpRequest (XHR): stránka normálně načte, uživatel klikne na HTML, funkce spustí službu přes XHR, dostane data, modifikuje původní HTML stránku

Same Origin Policy: JS kód může přistupovat pouze ke zdrojům na stejné doméně *(stejném originu)*
- důvod: ochrana před CSRF *(zneužití cookies)*, řešení: JSONP (GET), CORS

Scripting attacks: útočníci provádí akce se zlými side efekty

Cross-site Request Forgery: stránka / dokument obsahuje obrázek s akcí
-
- prevence: respektovat REST (vyžadovat POST), kontrola HTTP referer

Cross-site Scripting Attack (XSS): útočník vloží škodlivý kód na webovou stránku
- uživatel navštíví stránku, spustí se kód *(injection v Twitteru)*
- prevence: escapování uživatelského vstupu

Cross Origin Resource Sharing (CORS): řešení pro stále narůstající počet mashup aplikací
- povoluje HTTP požadavky napříč více stránkami, hlavičky Origin, Access-Control-Allow-Origin: *
- Access-Control-Allow-Methods, …-Allow-Headers, Access-Control-Max-Age

Simple request: GET / POST *(formuláře)*, specifické hlavičky a Content-Type
Ostatní požadavky: pomocí OPTIONS se provede **pre-flight request**, nesplněn => požadavek neodešlu

JSONP: založen na JSON *(objekt, pole, vložení, podpora v JS)*, využijeme element **script** a atribut **src**
- v URL: http://someurl.org/json_data?callback=load, stránka vrátí load({"people":"xy"})
- workaround pro same-origin-policy, nepoužívá se XHR, funguje jen pro GET

Bezpečnost (koncepty): na úrovni přenosu / zpráv
- autentizace: ověření klientovy identity *(OAuth),* autorizace: ověření přístupu ke zdroji *(OpenID)*
- **důvěrnost** zprávy: udržení v tajnosti *(TLS),* **integrita** zprávy: nedojde ke změně
- non-repudiation: nezpochybnitelnost akcí *(bankovní transakci jsem opravdu provedl já)*

Přístup k datům: **GET**: basic, digest, **third-party, API**: basic, digest, OpenID, OAuth

HTTP Basic: **WWW-Authenticate**: Basic realm="PrivateArea"
- klient použije **Authorization**: Basic *<base64_token>*, Realm identifikuje "oblast" na úrovni serveru
- username:password **zakódováno** v base64 => není bezpečné pro použití bez TLS

HTTP Digest: **WWW-Authenticate**: Digest realm="…", nonce, …
- předá jednorázový **nonce** a hash algoritmus, kvalita ochrany
- klient zahashuje s pomocí svého username, hesla, quality of protection, pošle s dalším požadavkem

TLS handshake: zaručí dohodnutí na šifrování
- **autentizace**: ověření identity serveru (certifikát), **integrita**: každá zpráva je podepsána pomocí MAC
- **postup**: SYN, SYN ACK, ACK (klasicky TCP)
=> ClientHello: verze TLS, seznam ciphersuites, nastavení TLS
=> ServerHello: verze TLS, ciphersuite, certifikát
=> výměna RSA / Diffie-Hellman klíčů => kontrola integrity zpráv => zašifrovaně zpráva "Konec"

Server Name Information (SNI): informace o serveru *(doméně)*, kam se připojuji
- klient ji vkládá do ClientHello zprávy, využívá ji load balancer *(nedostane se k hlavičce HTTP Host)*

Výměna klíče:
- kdysi RSA (privátní a veřejné klíče)
- klient vygeneruje symetrický klíč, zašifruje veřejným klíčem serveru, pošle, server ho pak dešifruje
- **nevýhoda**: pro autentizaci i šifrování stejný klíč
- **řešení**: Diffie-Hellman: klient a server se domluví na společném shared secret i bez komunikace
- nedochází k nebezpečí kompromitace minulých komunikací

Application Level Protocol Negotiation (ALPN): informace o dalším protokolu, který je pod TLS
- pomůže jinak dekódovat textové / binární protokoly

TLS na Proxy serverech:
- **TLS Offloading**: TLS mezi klientem a load balancerem, mezi LB a mým serverem nezabezpečené *(pokud je například LB vstup do mé privátní sítě, uvnitř sítě můžou lidi prozkoumávat zprávy)*
- **TLS Bridging**: TLS mezi klientem a LB, TLS mezi LB a serverem *(jedině proxy může prozkoumávat zprávy, organizačně náročnější – nejde ladit uvnitř systému)*
- **TLS pass-through** (end-to-end): TLS skrze LB pouze prochází *(proxy nemůže zkoumat zprávy)*

Proxy: používá TLS offloading / TLS bridging nebo TLS pass-through s pomocí Server Name Indication

Autorizace – tokeny: alternativa k standardnímu session stateful přihlašování
- stateless varianta, access token *(přístup ke zdroji)* / refresh token *(obnovení access tokenu)*

Best practices: access i refresh token je vázaný na IP adresu
- platnost access tokenu je 5 minut *(pak refresh token)*, refresh token je jednorázový
- přihlášení nebo obnovení tokenu zneplatní všechny předchozí

JSON Web Token (JWT):
- specifikace jedné konkrétní podoby access tokenu, token obsahuje i základní informace k autorizaci
- bezpečný, hlavička Authorization: **Bearer**, <header>.<payload>.<signature>

Hlavička: typ tokenu a hash algoritmus
Payload: claims (informace) – standardní (IANA claims) / custom
Signature: podpis pomocí secretu

OAuth 2.0 – motivace:
- Cloud Computing: Software as a Service, umožníme dalším aplikacím používat naše API
- podpora pro client-side / server-side / native
- uživatel zpřístupní data aplikacím třetí strany, **aniž** by těmto datům třetí strany dala heslo
- uživatel může kdykoliv odebrat (revoke) přístup

Pojmy: klient *(aplikace třetí strany, přistupuje ke zdrojům)*, resource owner *(vlastní zdroje)*, authorization / token endpoint, resource server

Ukázková implementace: Google OAuth 2.0
- autorizační kód *(k zisku tokenu)* vs access token, hlavička Authorization: OAuth xyz

Client-side přístup: autorizační server -> kód -> access token
- Javascript, nepoužívá se refresh token, pozor na CORS *(Google Console -> callback_url)*

Server-side (backend) přístup: autorizační server -> kód -> access i refresh token
- pokud access token vyprší, proběhne obnovení s refresh token
- kód je viditelný, ale k použití potřebuji **client_secret**

OpenID: pokročilá možnost autentizace do různých služeb, **silné** ověření identity 3. stranou

Sekvence interakcí:
1. požadavek aplikace k přihlášení => uživatel se přihlásí
2. objevení služby **(discovery)** => eXtensible Resource Descriptor Sequence (**XRDS)** s inform. o službě
3. požadavek o autentizaci => přesměrování na login page poskytovatele => přihlášení => identita

OpenID Connect (OIDC): nadstavba nad OpenID pro kompatibilitu s OAuth 2.0
- autorizační server pro kontrolu identity, endpointy, **API-friendly**, nativní integrace s Oauth 2.0

Sekvence interakcí: uživatel požádá o přístup, aplikace ho přesměruje na autorizační endpoint OIDC providera => po autorizaci se zavolá callback, aplikace požádá o access token + informaci o uživateli

WebSocket: umožňuje streamovat požadavek, samostatný protokol (nezávislý na HTTP)

Polling: čtení dat ze serveru v intervalu *(zatěžuje aplikaci)*
Pushing: updaty ze serveru
- long polling: server pozdrží požadavek, odpoví a konec *(vhodné pro dlouhý požadavek na API)*
- streaming: server posílá aktualizace bez zavření socketu *(posílání zpráv / souboru)*

HTTP Streaming:
- chunked response pomocí Transfer-Encoding: chunked
- pak vždy pošlu velikost (hex) dat a data, konec = 0

Server Sent Events: *(id: 12345\n data: řádek 1\n řádek 2\n\n)*
- API, které funguje přímo v prohlížeči, rozhraní **EventSource**, handlery open/message/error
- typ dat: text/event-stream, cache-control: no-cache
- pokud zvolím vhodně identifikátory, můžu využít **Last-Event-ID** při ztrátě spojení k znovuvytvoření

Streams API: umožní streamovat data přes Fetch API, ReadableStream *(getReader)*
Cross-document messaging: d.getElementById("iframe").contentWindow.postMessage("…")

WebSocket (wss://): nový protokol, běží nad TCP, oboustranný
- respektuje same origin policy / CORS / více server-side endpointů

Connection Upgrade:
- klient pošle hlavičku Connection: Upgrade a hlavičku Upgrade: websocket
- server přijme požadavek, odpoví 101 Switching Protocols
- následně Sec-WebSocket-Key/Origin/Protocol
- tento klíč se následně použije pro komunikaci ve WS

Data Framing: data se posílají v TCP packetech pomocí payloadu, zpráva je rozdělena do framů
- není zde HTTP = žádný HTTP overhead, jak klient, tak server zde můžou zapisovat
- **FIN**: je frame poslední / opcode *(text/bin),* klient má k dispozici API (metoda send)
- maskovací bit / klíč: maskuje (nešifruje) => ochrana proti **proxy cache poisoning**

Head-of-line blocking: zprávy se zpracovávají podle ID, 1 dlouhá zpráva může zablokovat systém
- řešení: rozdělení zpráv na menší části, nezahlcovat *(velikost bufferu, pošlu až když je prázdný)*

Proxy cache poisoning (útok)*:*
- útočník zapíše do Websocketu data jako HTTP – dokáže takto obejít same origin policy
- proxy může pak interpretovat toto jako HTTP, změnit obsah cache a zanést tam **poison** (jed)

Good practice:
- WebSocketu nastavíme *(my)* timeout 30 sekund *(neprojdou data = za 30 sekund se vytimeoutuje)*
- parametr Keep-Alive: přepoužití TCP socketů pro různá spojení (nastavení HTTP spojení)

Establish HTTP/2 připojení:
1) pomocí TCP a ALPN (HTTP/2)
2) Upgrade: h2c, Connection: Upgrade *(nepoužívá se – není zabezpečené)*

Binary Framing: data jsou přenášena v rámcích, zpráva je rozdělena do rámců, ty se můžou prokládat
Obsah framu:
- délka, typ (HEADERS, DATA, PRIORITY, PUSH, GOAWAY – *vyčerpán počet identifikátorů*)
- flags *(8 bits),* identifikátor streamu *(31 bitů)*

Stream: každý rámec je zařazen do streamu, pomocí streamů se propojují request/response
Životní cyklus streamu: idle => open => closed

Message: zpráva, kterou posílám, rozdělím do rámců

Request / Response Multiplexing: umožňuje odesílat a přijímat více požadavků
- oproti HTTP/1.1 zde nehrozí HTTP head-of-line *(pořád ale může nastat na úrovni TCP)*

Prioritizace streamu: nejprve potřebuji načíst HTML, pak CSS, obrázky, JS
- každý stream má váhu od 1 do 256 + podle hierarchie (*váhy se pak můžou nasčítat podle rodičů)*

Flow Control: už na úrovni TCP, stejný mechanismus na úrovni HTTP/2
- když nestíhám zpracovávat data, hromadí se v bufferu, pošlu zprávu na server a ten omezí rychlost
- princip: window size = počet dat, které lze odeslat; něco odešlu = zmenší se mi počet dat; 0 = stop
- obnova: WINDOW_UPDATE s novou délkou okna

Např. streamování videa = při pozastavení zastavit přenos dat / rozmazaný obrázek, další zdroje

Server Push: umožní poslat dodatečné zdroje v jednom požadavku
- obdoba **resource inlining** (zdroj je uživateli odeslán v HTML/CSS zdroji)
- výhoda: cache, reusing, multiplexing, možnost odmítnutí klientem, typ **PUSH_PROMISE**

Header Compression: hlavičky jsou komprimovány Huffmanovým kódováním v HPACK formátu

HTTP/3: řeší problém TCP head-of-line blocking, používá protokol QUIC založený na UDP