NI-AM2

Úvod

Web 2.0 - Read-Write, Programmable, Realtime, Social Web

Programmable - Aplikace poskytují data i funkcionalitu, firmy vystavují API

Scope

- Cloud Architektura - IaaS, PaaS, Microservices, Kontejnery, Docker, Kubernetees
- Pokročilé HTTP - Same origin, Cross-origin, OAuth, JWT, Open ID, Realtime Web, HTTP/2, HTTP performance

Asynchronní I/O

Modely pro komunikaci s Aplikačním serverem

- Blokující (synchroní) I/O
- Neblokující (asynchroní) I/O

Asynchorní I/O

- concurrent programming (tasky se překrývají), ne paralelní
- single-thread, single process
- cooperative (non-preemptive) multitasking - sám si rozhodnu jak dlouho poběžím
- tasky běží v **event loopu**, při čekání na resolve/reject můžu nechat pracovat jiného

JavaScript

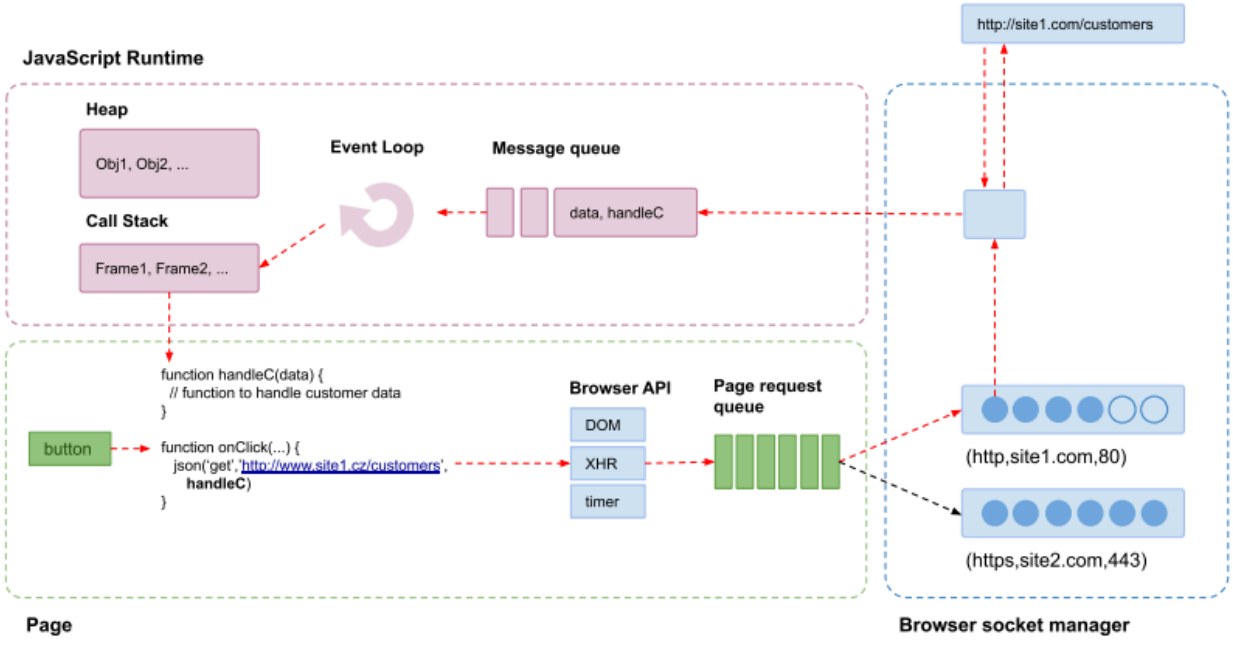
- **Web 2.0** - AJAX, dynamické stránky, asynchronní volání na server (XHR)
- **XHR** (XMLHttpRequest) - JS třída pro asynchronní posílání HTTP requestů
- ECMAScript - Standard pro jazyk
- funkce jsou objekty - dají se posílat, vracet, přiřazovat proměnným, ukládat
- anonymní funkce - inline funkce
- **this** si každá funkce definuje jinak, řešení: var that = this; that.age++
- **Arrow function** - kratší zápis funkcí: () => “hello”, nemá problémové vlastní this
- **Closures** - funkce, která referencuje proměnné mimo svoje tělo

Promise Object

- reprezentuje výsledek asynchronní operace (proxy k budoucí hodnotě)
- **async** - funkce, která vrátí Promise
- **await** - počkat na Promise

JS Runtime

- **Stack** - obsahuje parametry volaných fcí, lokální proměnné, …
- **Heap** - objekty na haldě
- **Queue** - seznam zpráv ke zpracování (data a callbacky)
- **Event Loop** - zpracovává zprávy z Queue
- Prohlížeč vytvoří do Queue novou zprávu při zavolání Eventu na EventListener
- Funkce/Zpráva je zpracovaná celá než se začne s další
- iframe a Web workers mají vlastní runtime
- Runtimy mezi sebou komunikují přes **postMessage** (naslouchání message event)



Web Workers

- Kód na worker thread, má Event loop, komunikuje přes posílání messages
- Může vše kromě manipulace s **DOM**
- Thread-safe
- **Dedicated workers** - přístupné ze skriptu, který je vytvořil
- **Shared workers** - přístupné z více skriptů (iframes, workers, …)

```
// main.js
var myWorker = new Worker('worker.js');

something.onchange = function() {
  myWorker.postMessage([value1,value2]);
}

// worker.js
onmessage = function(e) {
  var workerResult = 'Result: ' + (e.data[0] * e.data[1]);
  postMessage(workerResult);
}

// ... and terminate
myWorker.terminate()
```

Node.js

- Event-driven I/O Framework, všechna volání jsou asynchronní
- Requesty zpracovává worker thread (nemusíme řešit concurrency)
- Každé I/O je Event - R/W do souboru/socketu
- **Event Loop**
    - timers - spustí FIFO callbacky z setTimeout() a setInterval()
    - I/O callbacks - spustí všechny I/O callbacky kromě close callbacků
    - idle/prepare - interní využití
    - poll - získání nových I/O eventů
    - check - zavolá setImmediate() callbacky
    - close callbacks - spustí close callbacky (socket.on('close', …) atd…)

Google Apps Script

- JS cloud scripting napříč Google produkty

Cloud Architectures

Outsource aplikační infrastruktury

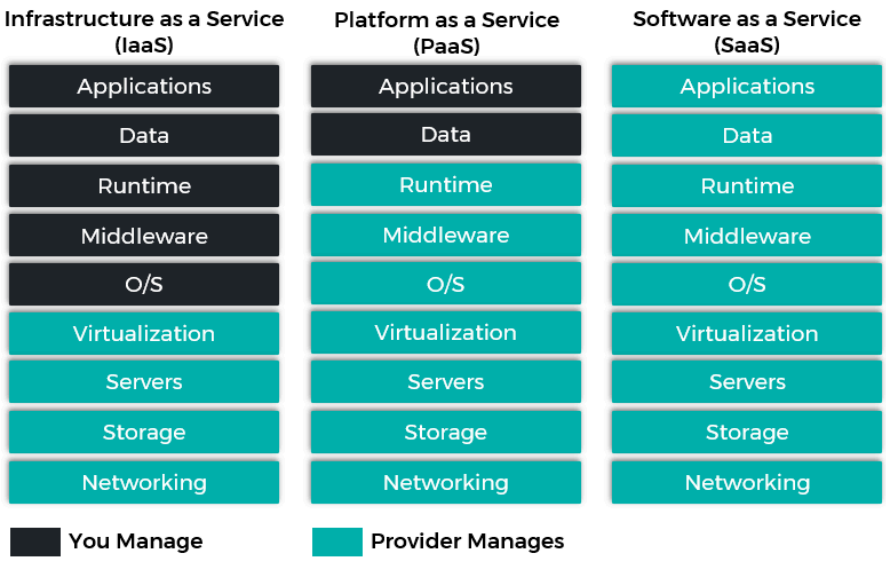
- spolehlivost, dostupnost, cena, elasticita
- CapEx - zaplatím jednorázově
- OpEx - zaplatím za maintenance (Cloudové řešení)

Koncepty Cloudu

- **On-demand** - zdroje jsou alokovány když jsou potřeba, automaticky
- **Broad network access** - přístupné z netu
- **Resource pooling** - výpočetní zdroje jsou užívány více uživateli (multitenancy)
- **Škálovatelnost a Elasticita** - infrastruktura se natahuje
- **Measured service** - monitoring
- **Pay-per-use** - zaplatíš za to co použiješ

Druhy služeb (modely)

- **IaaS** - Load balancer, monitoring, OpenNebula, Amazon EC2, Google Cloud, …
- **PaaS** - Kontejnery, mikroservices, serverless, OS, Runtimy, AWS, Kubernetes …
- **SaaS** - SW pro end-user, přístup přes web, Dropbox, Trello, …
- Public/Private/Hybrid Cloud



Infrastructure as a Service

Multitenancy

- Architektonický přístup pro sdílení zdroje mezi více uživatele
- **Shared everything** - typciké pro SaaS, izolace, sdílená DB tabulka, …
- **Shared infrastructure**
    - VM - virtualizace, hypervisor řeší izolaci
    - OS virtualizace - izolaci řeší OS, virtual environment

Virtual Cloud Network

- Private network in a single region
- Vlastní blok IPV4 CIDR
- Vlastní podsítě (Private/Public)
- Public podsítě můžou do internetu
- Route Tables - pro směrování do/z podsítí do/z internetu
- Peering
    - Local - propojení VCN v rámci regionu
    - Remote - propojení VCN napříč regiony
    - Připojení k on-premise datacentru - potřeba secure VPN

Výpočetní instance

- Shape - množství CPU a RAM na instanci
- VM - multi-tenant, virtualizace Bare Metal na menší VMs
- Bare Metal - single-tenant, celé železo pro vás
- Dedicated VM Hosts - single-tenant, VM instance běží na dedikovaném serveru
- **Autoscaling** - automatické škálování instance

Image

- Šablona s virtuálním diskem s OS a dalším SW, knihovnami, …
- Uložena na Boot volume, např. CentOS, Ubuntu, …

Load Balancer

- Health check, Round-robin, TCP, HTTP, WebSocket, SSL, sticky sessions, vysoká dostupnost

Storage

- **Object Storage**
    - Nestrukturovaná data (obrázky, logy, zálohy), data jako objekty
    - API přes HTTP
    - Namespace - top-level container pro všechny buckety a objekty (1 tenancy 1 namespace)
    - Bucket - container pro ukládání objektů, unikátní název v rámci tenanta
        - Hot bucket - okamžitý přístup
        - Cold bucket - málo používaná data, je třeba obnova (Time to first Byte)
    - /n/<namespace>/b/<bucket>/o/<object_name>
- **Block Storage**
    - Lokální SSD, obvykle bez RAIDu
    - Data uložena v bloku SSD
    - Přístup jako k disku, Network File Server endpoint
    - sudo mount 10.0.0.6/example/path /mnt/mountPointA

Infrastructure as Code

- Aplikační env, version control, team development, scripting, …
- Terraform - abstrakce datacentra

Cloud Native

Cloud Native Computing Foundation

- budují udržitelný ekosystém pro cloud native SW, součást Linux Foundation

Cloud Native - škálovatelné aplikace na moderním cloud prostředí

- kontejnery, mikroslužby, service mashes
- Výhody: systémy jsou méně provázané (loosely coupled), pozorovatelné, spravovatelné, automatizace
- Trail Map - overview pro začínající podniky do cloud native
    - kontejnerizace, CI/CD, orchestrace, pozorování & analýza

Microservices

Aplikace jako nezávislé nasaditelné služby (mikroslužby).

Hlavní charakteristiky

- Loosely coupled - integrace přes interface
- Protokoly - HTTP, REST
- Nezávislé, nahraditelné - změna ovlivní jen deploy mikroservisy
- Organized around capabilities - accounting, billing, recommendation, …
- Všestrannost implementace - jazyky, DB, vyber si

Kontejnery

Rozdíl oproti VM - kontejnery jsou izolované, ale sdílí OS, knihovny, binárky

Docker

- Postaven na Linux namespacu
- Build, commit and share images
- image build popsán v Dockerfile
- Hromada dostupných základních imagů

Containerd

- Container Engine - dostane user input (CLI, API), stáhne image z registry, připraví metadata, pošle do Container Runtime
- Container Runtime - abstrakce syscallů nebo OS fcí pro běh kontejnerů, komunikuje s kernelem pro start kontejner procesů, používá *runc* a *container-shim*

Terminologie

- Image - FS stackované na sobě, immutable, beze stavu
- Container - Procesy běžící v izolovaném namespace ve FS poskytnutém z Image
- Container Engine/Runtime - Hlavní procesy poskytující kontejnerizaci na hostiteli
- Client - Aplikace (CLI) komunikující s Container Engine přes API
- Registry - Hostovaná služba obsahující repozitář Imagí, Docker Hub
- Swarm - Cluster docker enginů, jejich managování, orchestrace

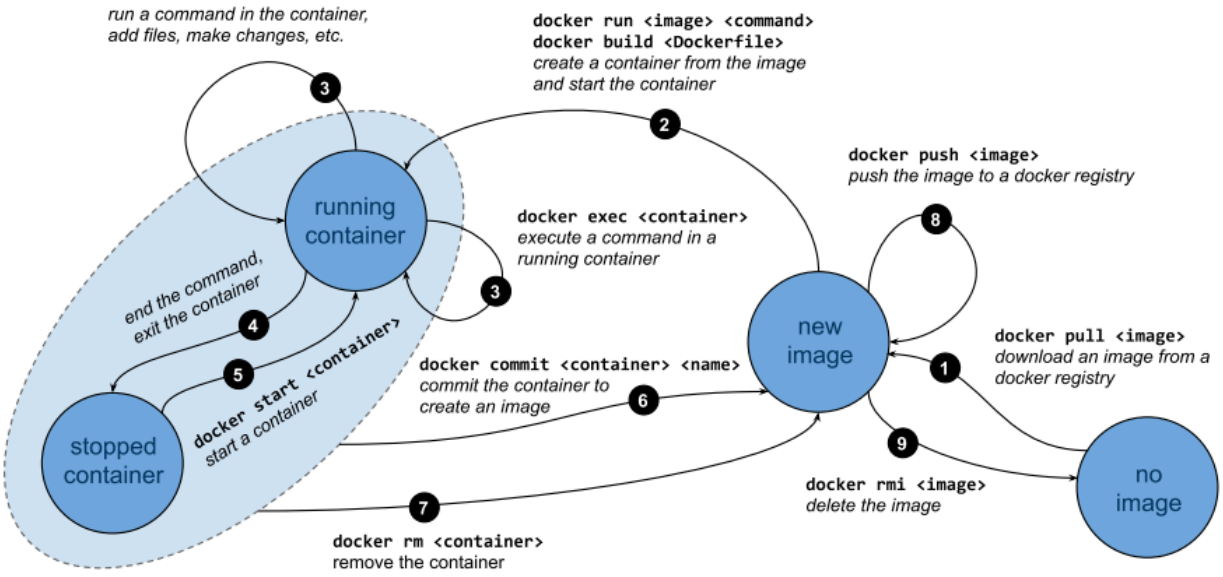
Kernel technologie pro kontejnerizaci - Control Groups, Union Mount, Namespaces

Linux namespace

- Izolace Linux procesů
- 7 namespaců - Mount, UTS, IPC, PID, Network, User, Cgroup (příkaz lsns)
    - net namespace - procesy mají vlastní network stack (interface, routing tabulky, sockety), komunikace s vnější sítí jde přes Virtual Ethernet Bridge
- appky v namespace si můžou povídat, ostatní ne

Container Image

- **OverlayFS** - FS služba, která implementuje **union mount** pro ostatní FS
    - vezme 2 složky, naskládá je na sebe jako vrstvy a poskytne unified view
    - funguje jen pro 2 vrstvy, pro víc vrstev referencuje hard linky
    - image layery jsou uloženy v /var/lib/docker/overlay/ ve složkách
    - pro efektivní využití disku
- **Dockerfile** - skript, který vytvoří novou image
    - každý řádek vytvoří zprostředkovávající layer
    - když to někde failne, obnoví se to od posledního stepu z cache



1: There is no image in the local store; you pull an image a remote registry.
2: You run a new container on top a specified image.
3: You modify the container by adding a library/content in it; you can also run a command in the container from the host.
4: You stop a running container.
5: You start a stopped container.
6: You commit the container and create a new image from it.
7: You remove the container.
8: You push the image to the remote registry.
9: You can remove the image from the local store.

Základní Docker příkazy

- docker version - verze
- docker search <image> - prohledá registry
- docker pull <image[:version]> - stáhne image, bez version stáhne nejnovější
- docker images - list lokálních imagí
- docker run -it <image[:version]> <command> - spustí image a command v něm
- docker ps [-as] - list všech běžících kontejnerů [a - stopped, s - velikost]
- docker rm <container> - smaže kontejner
- docker rmi <image> smaže image
- docker commit <container> <name[:version]> - vytvoří image z kontejneru
- docker history <image> - historie image

Networking

- Docker má by default 3 sítě:
    - **bridge** - propojí kontejner s hostitelskou sítí
        - Docker vytvoří podsíť 172.17.0.0/16 a gateway do sítě
        - Všechny kontejnery v této síti spolu mohou komunikovat
    - **host** - všechny hostitelské síťové rozhraní jsou z kontejneru přístupné
    - **none** - kontejner bude mít vlastní síť, žádné rozhraní není konfigurované
    - lze vytvářet vlastní, budou spolu komunikovat v rámci sítě, ale budou izolované od hostitelské sítě
- Linkování kontejnerů
    - $ docker run -dp 6379:6379 --name redis-server redis
    - $ docker run -it --link redis-server:redis --name redisclient1 redis sh
    - $ docker run -it -p 8080:8888 --link redis-server am2-04

- Základní příkazy
    - docker network ls
    - docker network inspect <networkId>
    - docker network create --driver bridge <networkName>
    - docker run -it --network=<networkName> ubuntu bin/bash

Data volumes

- Perzistentní úložiště i po odstranění kontejneru, přepoužitelné více kontejnery
- Vytvořit volume - docker run -d -v /webapp training/webapp …

- Mount host složky jako volume - docker run -d -v /src/webapp:/webapp …

Kubernetes

Automatizace deploymentu, škálování, správy containerized aplikací napříč více nody.

Deployment spec - podobné Dockerfile, Kubernetes přečte a podle toho koná.

Funkce

- Automatic binpacking - automatické umisťování containerů do nodů podle požadavků
- Horizontal scaling - přes UI nebo dle CPU usage (přidávání nodů)
- Automatizované rollouts, rollbacks - monitoring, rollback když se něco pokazí
- Storage orchestration - automaticky mountuje storage
- Self-healing - restartuje kontejnery, rescheduluje když umře node, zabíjí kontejnery které neodpovídají, …
- Service discovery - přiřazuje IP kontejnerům a DNS pro skupinu kontejnerů
- Load balancing - load-balancuje mezi nimi

Control Plane

- Centrum ovládání Kubernetes
- Ovládání clusteru, scheduling, zapínání podů, …
- kube-apiserver - vystavuje Kubernetes API do FE pro Control Plane
- etcd - uložiště pro všechna data clusteru (high-available)
- kube-scheduler - scheduler, vybírá nody Podům, zohledňuje zdrojové požadavky a constrainty/policy/specifikace, …
- kube-controller-manager - spravuje controllery pro udržení chtěného stavu celého clusteru
    - Node controller - reaguje na stav nodů
    - Job controller - vytváří Pod pro one-off tasky
    - Endpoints controller - řeší endpointy
- cloud-controller-manager - integrace s cloud službami
    - Nastavuje VLAN, load balancery, …

Node

- běží na něm Kubernetes Runtime Environment, na tom běží Pod
- kubelet - agent, který běží na každém nodu, zajišťuje, že kontejnery běží v Podu
- kube-proxy - stará se o networking na nodu

- Container runtime - stará se o běžící kontejnery, implementace Kubernetes CRI

Pod

- skupina kontejnerů sdílející namespace, storage a network zdroje
- na Podu běží 1 instance nějaké aplikace
- vytváří se na základě **workload** resources, ne manuálně
- většinou 1 kontejner, ale může jich být víc co jsou spolu úzce provázány

Workloads (zdroj)

- jsou aplikace které běží v Kubernetes
- vytváří Pody
- Předdefinované workloady pro správu Podů:
    - Deployment - spravuje stateless aplikace
    - StatefulSet - zajišťuje stavovost
    - DaemonSet - spustí appku na každém nodu (kopie Podu)
    - Job/CroneJob - skript, třeba DB schéma

Networking

- Kontejnery v rámci Podu spolu komunikují přes loopback
- Cluster networking poskytuje komunikaci napříč Pody
- Service resource - abstrakní způsob vystavení aplikace běžící v Podu
    - každý pod je součástí nějaké service
    - vytvoří se Service object, který targetuje konkrétní protokol třeba

minikube - lokální VM, běží na něm master node, umožňuje testování, demo na 1 stroji

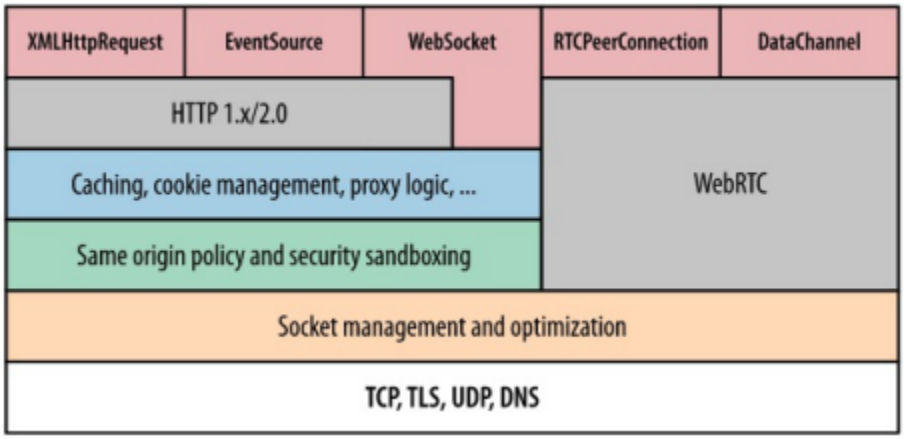
kubectl - CLI do Kubernetes clusteru

1. *create* `hello-node` *app in node.js and test it [see server.js]*
   `node server.js`
2. *create docker image for the app [see Dockerfile]*
   `docker build -t hello-node:v1 .`
3. *deploy the app to Kubernetes by using* `kubectl`
   `kubectl run hello-node --image=hello-node:v1 --port=8080`
4. *Expose the app as a load balancer service.*
   `kubectl expose deployment hello-node --type=LoadBalancer`
5. *Explore the app in minikube dashboard.*
   `minikube dashboard`
6. *Fire requests at the service and count them [see test.sh]*
   `./test.sh`.
7. *Change the number of replicas by using the dashboard or* `kubectl`.

Browser Networking

Prohlížeč

- Platforma pro delivery webových aplikací (rychle, efektivně, bezpečně)
- Komponenty - parsing, layout, kalkulace HTML CSS, JS execution, rendering pipeliny, **networking stack**
- když network čeká, blokují se ostatní kroky



Connection management

- přepoužívání socketů, prioritizace requestů, protokol, limitování připojení, …
- **socket manager** - organizace socketů v poolech

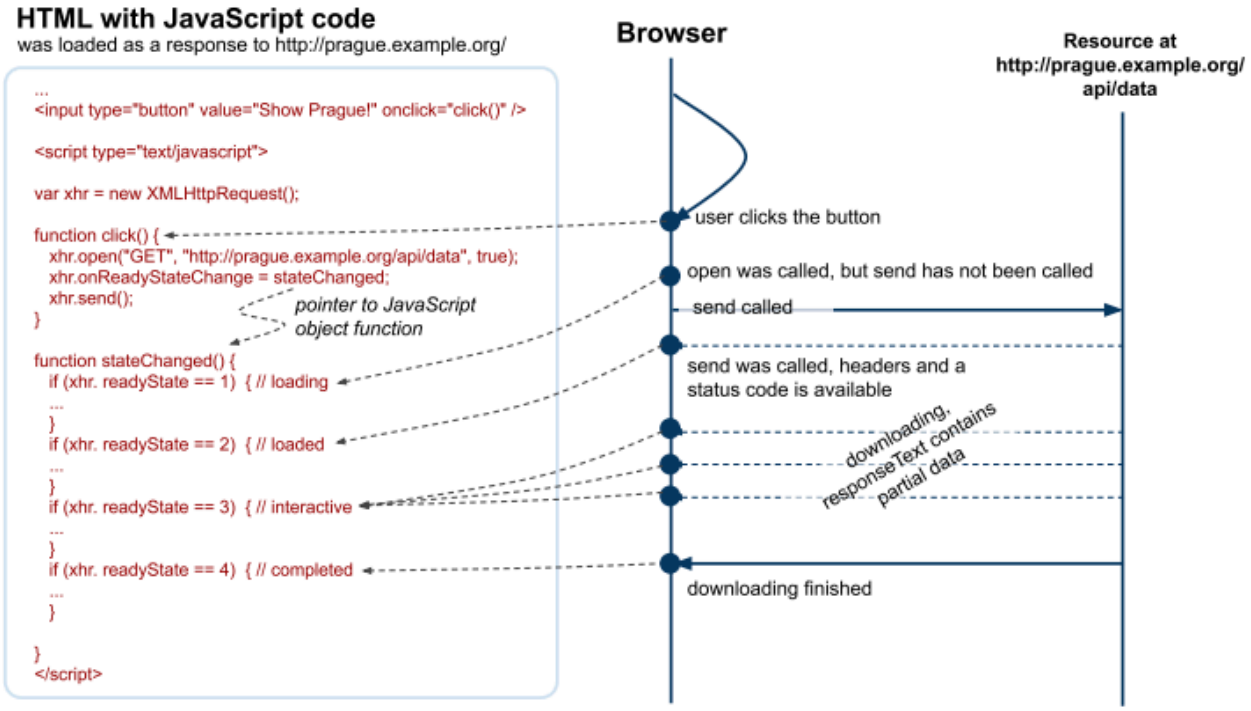
Security

- Aplikace nemohou iniciovat připojení k hostiteli (port scan, …)
- Connection limits - ochrana klienta i server před vyčerpáním zdrojů
- Request formatting - výchozí eventy jsou formátovány, protokoly, sémantika
- Response processing - dekódování response chrání uživatele
- TLS negotiation - TLS handshake, verifikace, certifikáty, user upozorněn
- **Same-origin policy** - JS může přistupovat jen k resource na stejné doméně (origin)
    - lze obejít přes JSON(JSONP) pro GET nebo CORS (cross-origin resource sharing protocol)
    - bez toho lze používat POSTy třeba do přihlášené banky na útočníkově webu
- **CSRF** - web útočníka má odkazy na citlivé zdroje pod přihlášením jiného webu
    - ochrana - REST (GET nesmí nic měnit), kontrolovat HTTP **referer** header
- **XSS** - web infikován kódem, klient omylem executne, útočník třeba získá cookie

Mashup - mashup více aplikací, používá API, Google mapy

XMLHttpRequest (XHR)

- Rozhraní pro HTTP v JS
- základ pro AJAX (asynchronní JS a XML)
- click -> trigger event -> invoke XHR (same-origin policy, cross-origin policy) -> data
- **XHR object**
    - open [method, url, asynch, user, pass] - otevře request s parametry
    - onReadyStateChange - zavolá se, když se změní readyState
    - send, abort - odešle nebo zruší request
    - status, statusText - HTTP status code a status text
    - responseText, responseXML - response, může být ve formátu DOM
    - onload - listener pro server push



CORS

- Čím dál více Mashupů, weby komunikují cross-origin
- Vždy používat Origin v HTTP headeru
    - Access-Control-Allow-Origin - které originy jsou OK
- **Preflight** - Pošlu OPTIONS před GETem pro zjištění, jestli jsem OK (browser posílá preflight pro každé XHR), cachuje se dle Access-Control-Max-Age

JSONP

- workaround pro Same-Origin policy
- wrappuje JSON response do funkce
- když url/json_data vrátí json, tak url/json_data?_callback=loadData vrátí loadData(json)
- nevyžaduje XHR
- vyžaduje podporu na straně webu, který voláme

Security

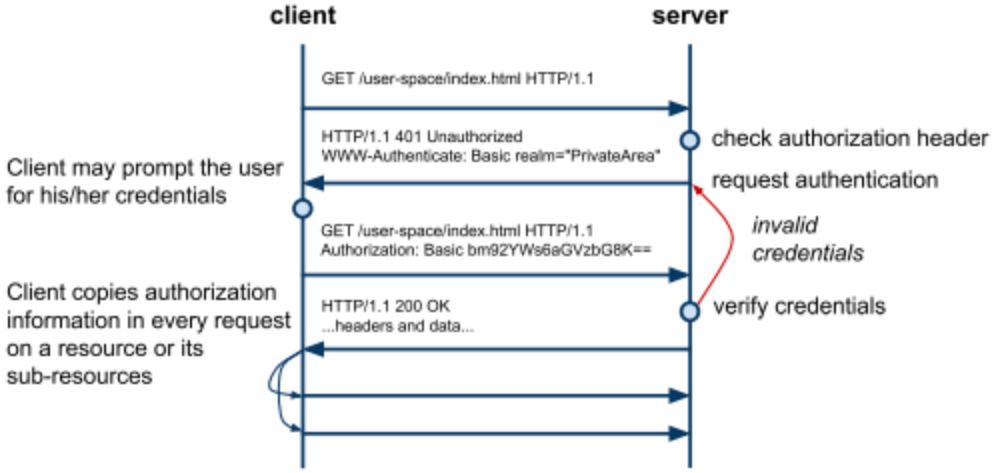
Message-Level security a Transport-Level security.

Zajišťuje

- Autentikaci - ověření identity
- Autorizaci - ověření oprávnění
- Message Confidentiality - šifrování zpráv
- Message Integrity - zprávy se nemění při přenosu (signed)
- Non-repudiation - ověření integrity a originu dat

Autentikace

- HTTP autentikace - WWW-Authenticate header, Authorization header



- **Basic Access Authentication**
  - Credentials - zakódovaná Base64, formát username:password, bez TLS čitelná
- **Digest Access Authentication**

1. *Client accesses a protected area*

```
> GET / HTTP/1.1
```

2. *Server requests authentication with* `WWW-Authenticate`

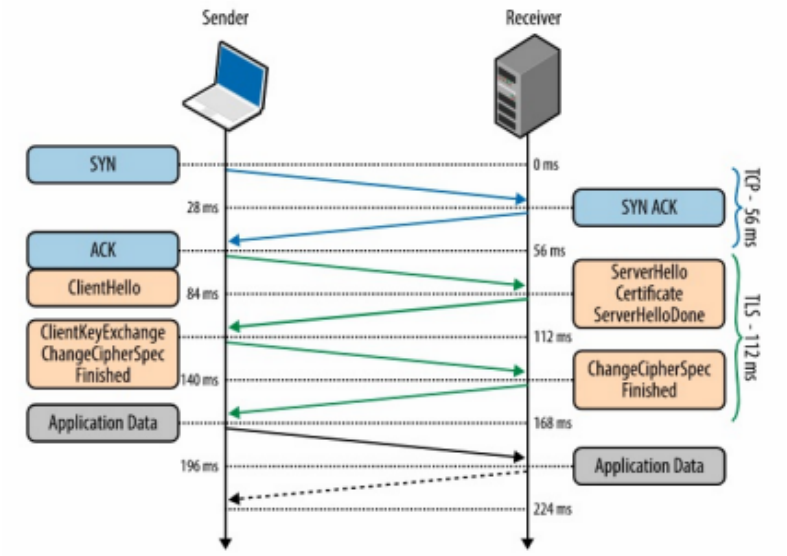
```
< HTTP/1.1 401 Unauthorized
< WWW-Authenticate: Digest realm="ProtectedArea",
  nonce="BbdQof3DBAA=a293ff3d724989371610f03015f2d23f3cd2c045",
  algorithm=MD5, domain="/", qop="auth"
```

3. *Client calculates a response hash by using the realm, his/her username, the password, and the quality of protection (QoP) and requests the resource with* `authorization` *header*

```
> GET / HTTP/1.1
> Authorization: Digest username="novak", realm="ProtectedArea",
  nonce="BbdQof3DBAA=a293ff3d724989371610f03015f2d23f3cd2c045", uri="/",
  algorithm=MD5, response="c4ea2293aeb318826d1e533f363efd90", qop=auth,
  nc=00000001, cnonce="531ee8ba7f2a8fd1"
```

Transport-Level Security (TLS)

- Šifrování - TLS handshake, dohodnutí na klíčích
- Autentikace - Chain of Trust, Certifikační autority, verifikace serveru i klienta
- Integrity - message framing mechanism, zprávy jsou podepsány Message Authentication Codem (MAC)
- **TLS handshake**



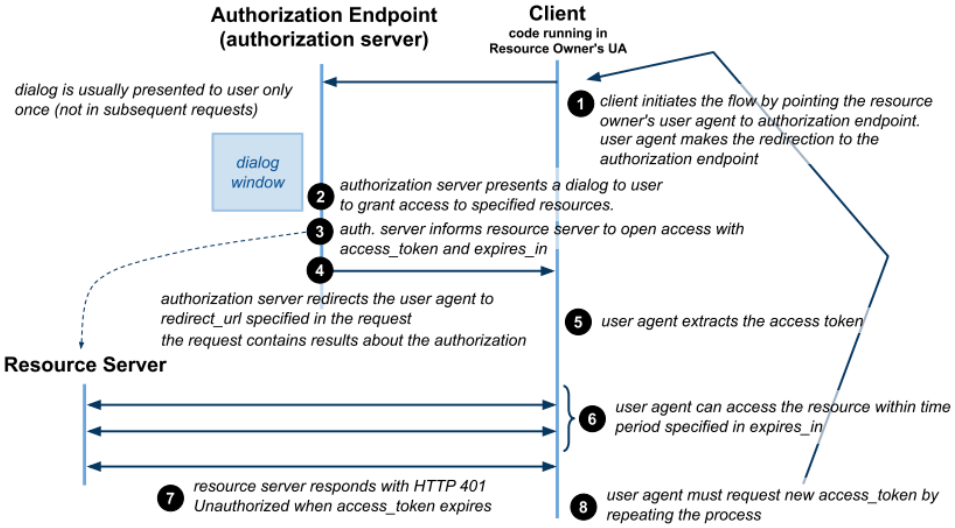
  - **Výměna klíčů** - RSA key exchange / Diffie-Hellman key exchange

JWT

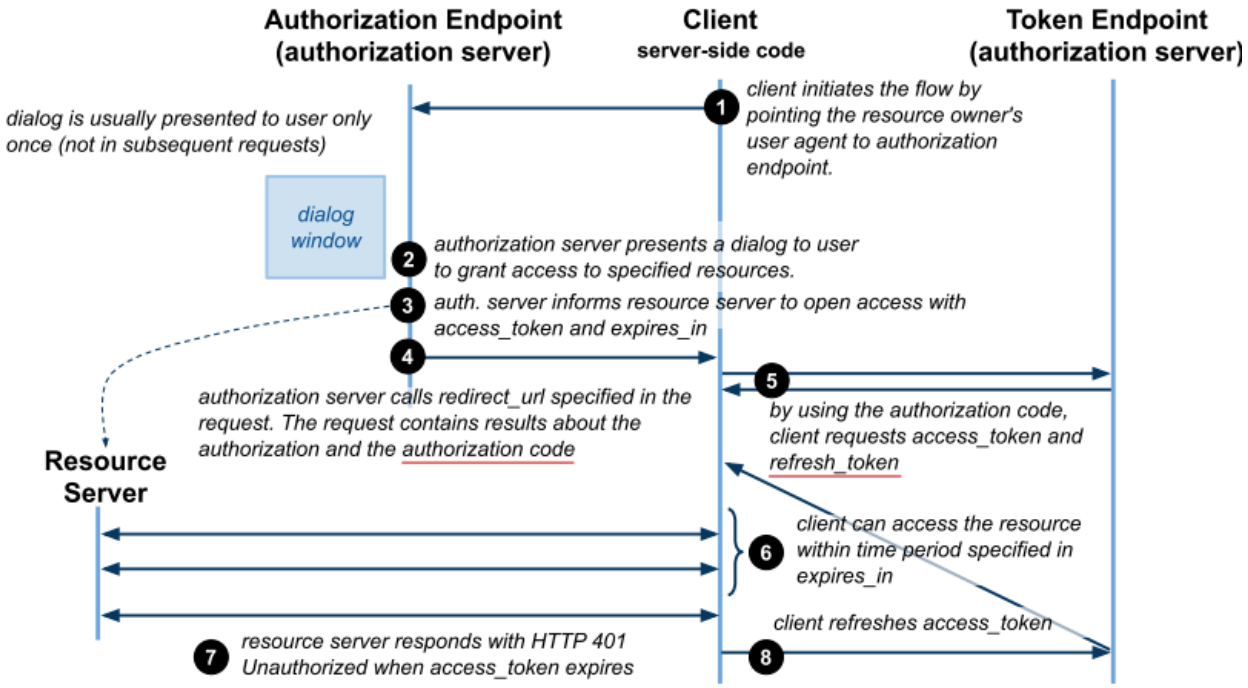
- ověřený, důvěryhodný a podepsaný JSON object
- kompaktní, šetrný, lze poslat všlijak (URL, POST, HTTP header), payload má vše
- **Autentikace** - používá se jako autorizační token, nejčastěji u Single-Sign On
- **Přenos informací** - token je podepsaný, takže integrita atd…

- **Struktura** - <header>.<payload>.<signature>
    - header - typ JWT a hash algoritmus
    - payload - info o uživateli atd
    - signature - HMACSHA256(base64(header) . base64(payload), secret)
- JWT nesmí obsahovat citlivé údaje, je čitelný

OAuth 2.0

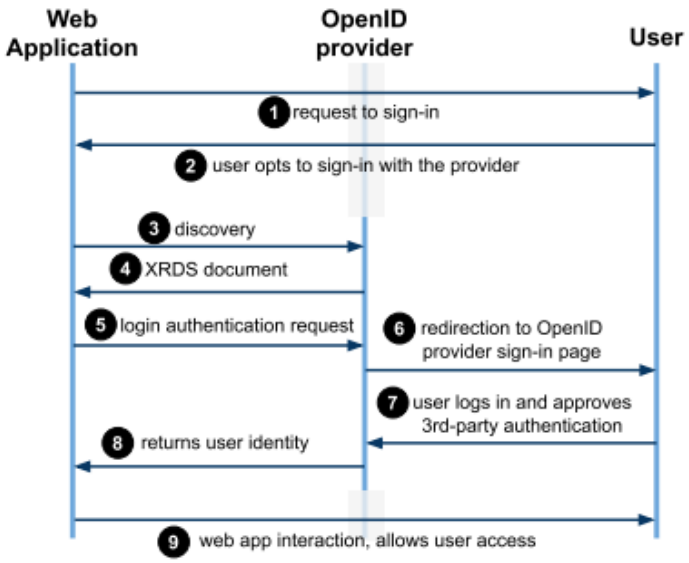
- uživatelé mohou přistupovat k aplikacím třetích stran bez sdílení jejich credentials
- mohou kdykoliv tento access revoknout
- iniciováno Googlem, Twitterm a Yahoo!
- **Terminologie**
    - Client - aplikace třetích stran, která přistupuje k uživatelskému zdroji
    - Resource Owner - uživatel
    - Authorization/Token Endpoints - endpointy poskytovány autorizačním serverem, přes který uživatel autorizuje své requesty
    - Resource Server - aplikace která má uživatelské zdroje, Google Contacts
    - Authorization Code - kód, který Client používá pro požádání o **access token** do resourcu
    - Access Token - kód, kterým Client přistupuje k Resource
    - 

- Příklad Google
    - Google Authorization Endpoint: https://accounts.google.com/o/oauth2/auth
    - query parametry
        - client_id - id klienta
        - redirect_uri - kam redirectnout po autentikaci
        - scope - identifikace resource, ke které bude umožněn přístup

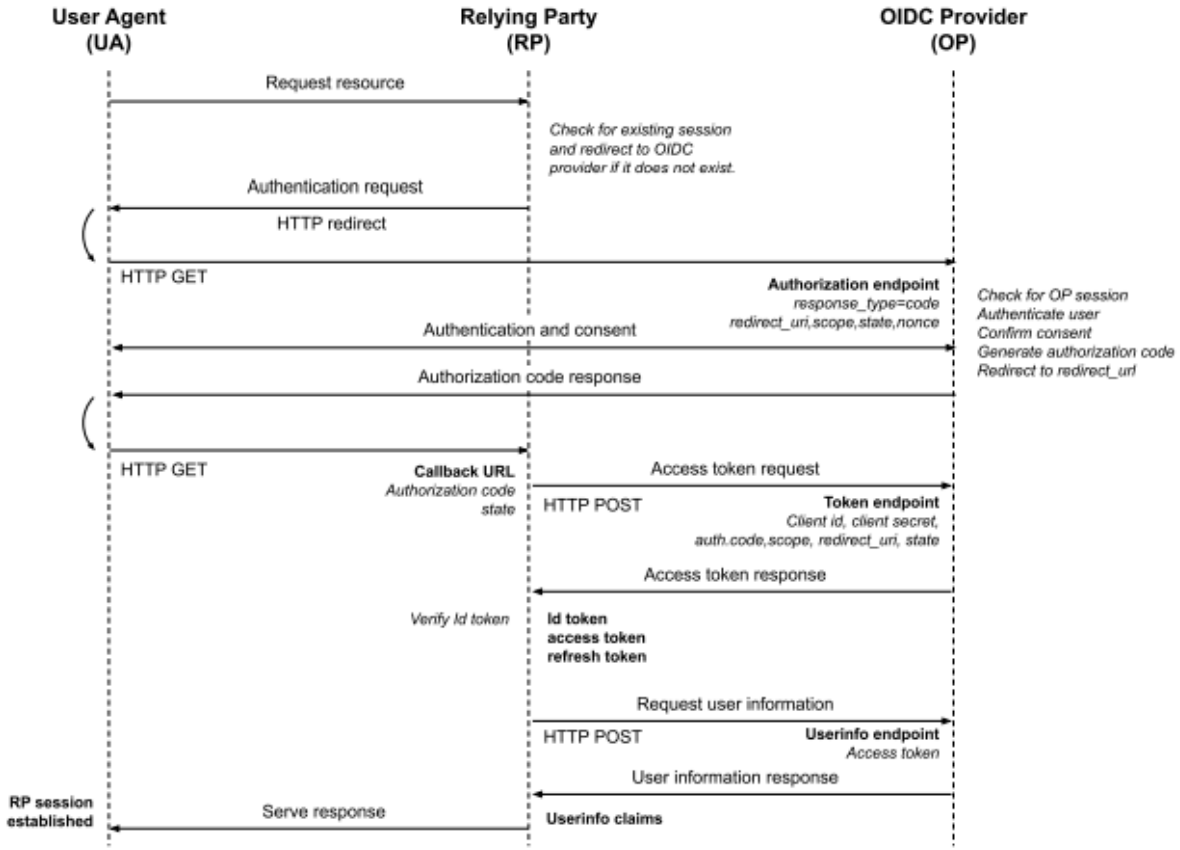
        - response_type - token nebo code
    - Klient parsuje **redirect_uri** URL v JS, získá **access_token** a **expires_in**
    - Po dobu expires_in lze přistupovat ke zdrojům na Resource Serveru
    - Během doby přístupu je povolený Access-Control-Allow-Origin
- To bylo client-side web app
- **Server-side web apps**
    - Autorizační server zavolá redirect_uri
    - klient získá **code**, tím si požádá o **access_token** přes POST na token endpoint (https://accounts.google.com/o/oauth2/token)
    - získá **access_token, expires_in** a **refresh_token**
    - refresh_tokenem se pak refreshne access_token znovu POSTem na token EP
- 

OpenID Protokol

- Uživatelé mají více účtů, více hesel
- OpenID umožní propojení těchto účtů pod jeden OpenID účet
- Jedná se o protokol, každý si může postavit vlastní OpenID provider



- XRDS - formát pro Discovery pro OpenID
- **OpenID Connect (OIDC)**
  - identity vrstva nad OAuth 2.0 (kombinace)
  - dělá to samé co OpenID + API friendly



Protocols for Realtime Web

Komunikace

- **HTTP request-response** - response se pošle po requestu, server neinicializuje
- **Polling** - jsou nová data? jsou nová data? Zatěžuje server
- **Pushing** - jakmile má server data připravená, pošle mi je v otevřeném spojení
    - long polling - server drží request a pak vrátí response, když jich drží moc, může být horší než Polling
    - streaming - server posílá změny bez uzavření socketu
- Chunked response - server posílá chunky, může tak poslat více setů dat v rámci jednoho spojení, které se ukončí po přijetí posledního chunku
    - chunky se musí rechunkovat do velkého chunku, Eventy jsou celé
    - client bufferuje chunky dokud z nich nebude schopný udělat response
    - HTTP streaming v prohlížečích - Server-sent events
- **Server-sent events (SSE)**
    - API pro HTTP streaming, využívá DOM eventy, zvládá chunky, same origin
    - **EventSource interface**
        - onopen, onmessage, onerror
        - readyState - CONNECTING, OPEN, CLOSED
    - response musí mít **content-type: text/event-stream**
    - každý řádek response musí být: **data: 1234\n**
    - reconnection je jednou za 3 vteřiny, lze změnit přes parametr **retry**

Streams API

- experimentální, JS API pro přístup ke streamu dat
- nemusí se čekat až přijdou všechny chunky, lze zpracovávat hned
- Fetch API, ReadableStream

Cross-Document messaging - vkládáním JS do iframů, server tam pak posílá messages

WebSocket

- nový protokol, vyžaduje podporu prohlížečů, webových serverů, proxy serverů
- obousměrná komunikace mezi klientem a serverem, low-latency, ne HTTP overhead
- Same-origin policy, CORS, podporuje více server endpointů

- ws / wss (TCP, TLS), staví nad TCP, nemusí být v prohlížeči
- Vyžaduje dedikované TCP spojení
- **Connection upgrade**
    - Klient pošle request na upgrade protokolu z HTTP na WS (header Upgrade)
    - Server vrátí response 101 Switching Protocols
    - Pak je navázán socket mezi klientem a serverem, oba do něj mohou R/W
- Data se posílají v TCP packetech (ve Frame), žádný HTTP header
- **Frame** - obsahuje header a payload
    - Payload - obsahuje část zprávy, zprávy se pak komponují do jedné
    - FIN - indikace, že je to poslední frame jedné zprávy
- **Head-of-line blocking** - requesty čekají na dokončení předchozích
    - velké zprávy na WS mohou způsobovat Head of line blocking u klienta
    - proto rozdělujeme zprávy do více framů (framy musí jít za sebou)
    - stále může blokovat, pokud jich bude hodně
    - je třeba rozdělit zprávu na malé zprávy
    - je třeba monitorovat buffer u klienta a posílat jen když je prázdný
- **WebSocket Browser API**
    - pro utilizaci WebSocketu, podpora v moderních prohlížečích
- Infrastruktura
    - Některé části sítě mohou blokovat WebSocket traffic - potřeba fallback strategy
    - Intermediaries mohou timeoutovat HTTP spojení - potřeba TLS tunelling

HTTP/2

Limity HTTP/1.x

- pro concurrency je třeba více HTTP spojení
- zbytečný traffic navíc - headery nejsou komprimované
- neefektivní prioritizace zdrojů

Cíle HTTP/2

- snížení latency, plný request/response multiplexing
- minimalizace overhead protokolu, efektivní komprese HTTP hlaviček
- podpora prioritizace a server pushe
- zachovat HTTP sémantiku (HTTP metody, URIs, headery, …), jen mění jak jsou **data** formátována a přenášena

Zahájení HTTP/2 spojení

- přes TLS a ALPN - klient pošle HTTP/2 protokol v TLS zprávě
- upgrade - klient pošle upgrade request z HTTP/1 na HTTP/2
- inicializace HTTP/2 spojení - klient zjistí info ohledně HTTP/2 na serveru z DNS a následně inicializuje spojení

HTTP/2

- **Komunikace**
  - **Binary framing** - framy mají binární formát
  - Komunikace **multiplexována** v rámci jednoho TCP spojení
    - HTTP/1 potřebuje pipelining (max 6 spojení), Head of line blocking
    - HTTP/2 má paralelní in-flight streamy, neblokují
  - Stream - obousměrný tok bytů, mohou nést více zpráv, mohou mít **prioritu**
  - Message - sekvence framů, mapují se na logický request/response message
  - Frame - každý frame má stream_id, kterému patří
  - nepoužívá znovu ty samé stream_id, jakmile id hitne 2^31, prohlížeč inicializuje nové TCP spojení
- **Stream prioritization**
  - prioritu má to, co něco blokuje
  - priorita posílání framů pro lepší výkon
  - každému streamu lze přiřadit váhu 1 - 256 nebo závislost na jiný stream
    - parent stream získá zdroje před dětmi
    - kdo má větší váhu získá větší resource
- **Flow Control**
  - Pause stream když klient pausne video třeba, kontroluje množství toku
- **Server push**
  - posílání více response na jeden request
  - klient se nemusí jednotlivě ptát na další zdroje, server ví co bude klient potřebova a tak mu to naservíruje
  - posílá se **Push Promise**, ne samotný zdroj, klient už to může mít v cache
  - snižuje latency z klientského dotazování
  - same-origin policy
  - ne vždy je to rychlejší - HTML size < Bandwidth-delay Product
- **Header compression**
  - komprese HTTP hlaviček přes HPACK formát (Huffman code)
  - posílají se jen ty hlavičky, které se změnily (static/dynamic tables)

Analýza HTTP/2

- nástroj nghttp - zobrazí framy
- Wireshark - traffic je šifrovaný, prohlížeče můžou poskytnout klíč v NSS Key Log Format

Nedostatky HTTP/2

- HTTP/2 závisí na TCP, to má svoje Head of line blocking
- Vyžaduje TLS handshake po TCP handshaku

HTTP/3

- Stejná sémantika napříč HTTP verzemi
- Nový transportní protokol QUIC (vychází z UDP), žádný head-of-line blocking, rychlejší navazování spojení, TLS 1.3 (HTTP/2 používá TLS 1.2+, HTTP/1 používá TLS/SSL a někdy ne)

Poznámky

- **Async** vždy vrátí Promise object
- Pokud chceme počkat na Promise resolve/reject, použijeme **Await**
- **Event Loop**
- **HTTP/2** - binární, HTTP/1 je textová