

# ÕV1: Loob hajusa arhitektuuriga rakendusi ja olemasolevatele teenustele klientrakendusi

**Soovituslik tölgendus sellele õpiväljundile:** loob veebiteenustele klientrakendusi, kasutades sünkroonseid ja asünkroonseid andmeedastusviise, valides neist lähteülesande lahendamiseks sobivaima;

## 1. VEEBITEENUSED

### Veebiteenus vs API

Programmeerimisliides ehk **API** on moodus, mis võimaldab kahel rakendusel omavahel andmeid vahetada, kusjuures need rakendused võivad olla kirjutatud täiesti erinevates keeltes.



Joonis 1: API kasutamine on nagu kaks rakendust vestleksid omavahel sõnumirakenduses

**Veebiteenus** on selline API, mis kasutab **HTTP protokolli** nagu veebiserveridki (Apache, Nginx) brauseritega suheldes (Firefox, Google Chrome), mistõttu saab neid ka brauseriga mõningasel määral (ainult GET pärtingute osas) kasutada.

Kõik veebiteenused on API-d, aga kõik API-d ei ole veebiteenused. Näiteks on olemas Windows API, mida kõik Windowsi töölauarakendused kasutavad ja mis võimaldab neil kasutada operatsioonisüsteemi sisse ehitatud funktsionaalsust, näiteks paluda operatsioonisüsteemil kuvada kasutajale mingu teade. Ja riistvaraga suhtlemiseks on eraldi API-d ([OpenGL](#), [Metal](#), [Direct3D](#)).

**Kaks köige levinumat viisi veebiteenuse tegemiseks on SOAP ja REST.**

Mis on nende erinevus? **SOAP on standard**, mis kirjeldab sõnumite vormingut, mida veebiteenus ja selle klient üksteisega vahetavad. **REST on aga kogumik mittekohustuslikke soovitusi** (*best practices*), kuidas hästikäituval rakendused *võiksid* andmeid üle veebi (see tähendab kasutades HTTP protokolli) vahetada ja igal veebiteenuse ehitajal on RESTist oma spetsiifiline nägemus, kuigi suures osas nad kattuvad.

Järgnevalt vaatleme kumbagi lähemalt.

## SOAP

SOAP (lühend on tületatud fraasist *Simple Object Access Protocol*, kuid versioonist 1.2 on öeldud, et SOAP ole enam akronüüm, vaid lihtsalt nimi) on natuke vanem veebiteenuste loomise viis, mis leidis varem laialdast kasutust. SOAP on väga ulatuslik, kuid keeruline standardite kogum. Nimelt Microsofti meeskond, kes kavandas SOAPI, tegi selle äärmiselt paindlikuks, et see võimaldaks suhtlemist nii privaatvõrkudes, internetis kui ka isegi elektronposti vahendusel.

## UDDI ja WSDL

SOAP-i esialgne versioon oli osa spetsifikatsionist, mis sisaldas ka Universal Description, Discovery and Integration (UDDI) nimelist standardit, mis oli mõeldud SOAP teenuste leidmiseks ja Web Services Description Language (WSDL), mis oli mõeldud SOAP teenuste dokumentatsiooniks. UDDI ei Levinud nii laialdaselt kui selle loojad lootsid ja 2006. aasta jaanuaris teatasid IBM, Microsoft ja SAP, et nad sulgevad oma avalikud UDDI serverid. 2007. aasta lõpus sulges ka UDDI-d määratlev rühm oma uksed ehk UDDI-t enam edasi ei arendantud ja 2010. aasta septembris teatas Microsoft, et nad eemaldavad UDDI teenused Windows Serveri operatsioonisüsteemi tulevastest versioonidest.

SOAP sätestab sisuliselt ümbriku ehk XML struktuuri veebiteenustega andmede vahetamiseks. Arhitektuur ise on loodud selleks, et aidata kaasa erinevate operatsioonide sooritamisele tarkvaraprogrammide vahel. Programmide vaheline suhtlus toimub tavaliselt XML-põhiste pärtingute ja HTTP-põhiste vastuste kaudu. Enamasti kasutatakse HTTP suhtlusprotokolli, kuid võib kasutada ka muid protokolle.

SOAP-sõnum sisaldab mõningaid kohustuslikke osi, nagu ENVELOPE, HEADER, BODY ja FAULT. ENVELOPE objekt määratleb XML-sõnumi taotluse alguse ja lõpu, HEADER

sisaldab kõiki serveri poolt töödeldavaid päiseelemente ja BODY sisaldab ülejäänu XML-objekti, mis moodustab taotluse. FAULT-objekti kasutatakse vigade käsitlemiseks.

## REST

REST-i (*Representational State Transfer*) on tavaliselt nimetatud pigem veebiteenuste arhitektuuristiiliks kui protokolliksi või standardiks. See tuleneb sellest, et REST ei määratle sõnumi sisu, vaid ainult teatud tingimusi, millele eeskujulik veebiteenus, mida on lihtne ja mugav kasutada, peaks vastama. Ka REST võimaldab suhtlust kahe tarkvaraprogrammi vahel: üks programm saab teiselt programmilt ressursse taotleda ja nendega manipuleerida. REST on üles ehitatud HTTP-protokollile, kasutades URL-ile sarnaseid viiteid ressurssidele, mida nimetatakse URI-deks (Uniform Resource Identifier) ja HTTP verbe nagu GET, POST, PUT ja DELETE, mis näitavad, millist tegevust klient ressursiga soovib teha. REST kasutab andmete edastamiseks kodeerimisformaate nagu XML, HTML või JSON. Kõige eelistatum on JSON, kuna see on kõige ühilduvam ja lihtsamini kasutatav.

REST on väga arendajasõbralik, sest selle kasutamine on palju lihtsam kui SOAP. Lisaks on REST vähem sõnarikkam ja kahe lõpp-punkti vahelisel suhtlemisel saadetakse vähem andmemahtu. REST lahendas SOAPI keerukuse probleemid ja nüüd on praktiliselt kõik avalikud API-d REST API-d.

```
POST /persons HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Host: celebrities.example.com
User-Agent: xh/0.14.1

{
    "name": "Vladimir Putin",
    "job": "president",
    "age": 69
}

HTTP/1.1 201 Created
Connection: Keep-Alive
Content-Length: 66
Content-Type: application/json; charset=UTF-8
Date: Thu, 17 Feb 2022 18:26:00 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.46
X-Powered-By: PHP/8.0.8
```

REST veebiteenusega saab suhelda kasutades HTTP protokolli. Antud joonisel esitab REST veebiteenus küsitud andmete hetkeoleku JSON formaadis.

## Mida tähendab RESTful?

Enamik API-sid maailmas on RESTful, mis tähendab, et nad järgivad suures osas teatud reeglite või öieti piirangute kogumit, mida tuntaksegi kui REST, mis on alates 2000.aastate algusest olnud de facto standard API-de arendamisel. De facto sellepärast, et ametlikult ei ole REST standard, vaid Roy Fieldingu poolt doktorikraadi väitekirjas kirja pandud parimate praktikate kirjeldus, millele on aegade jooksul lisandunud ka teisi häid tavasid.

## Kuidas valida SOAPI ja REST-i vahel

Mõnedel programmeerimiskeeltel (nt Java) on väga hea SOAP tugi ja selle kasutuselevõtt on lihtne. Muus keeles pole see nii lihtne (nt Javascriptis pole keelde sisse ehitatud tuge ja tuleb kasutada väliseid teeke, mis on kohmakas).

Samuti on REST väga palju rohkem levinud ja suvalisel arendajal on suure töenäosusega RESTist mingisugune arusaam juba olemas, mistõttu on üldjuhul REST parem valik.

Võrreldes RESTiga on SOAP kindlasti keerulisem ja kasutab rohkem andmemahtu, aga SOAPI pakub ka eeliseid:

- transpordist sõltumatu (REST vajab HTTP-d, sest kasutab HTTP verbe käskudena ja URL-e andmekollektsioonidele viitamiseks, aga SOAPI puhul on kogu info XMLi sees, mis on POST päringu keha sees, aga tegelikult pole vahet, mis meediumi vahendusel XML-ide vahetamine toimub, kasvõi tuvipostiga).
- töötab hästi hajutatud ettevõtluskeskkondades (REST API server vajab otseühendust kliendiga, aga SOAPIga pole vahet, mitmest kohast XML dokument enne serverini jõudmist läbi käib (kasvõi läbi pastebin.com-i))
- on standardiseeritud (kõik on standardiga paigas, kuidas implementeerida, aga RESTi puhul peab disainiotsuseid ise tegema ja guugeldama neid, süvenedes arendajate pikkadesse filosoofilistesse vätlustesse, kas nii või naa oleks parem. Just seetõttu ongi kõik REST veebiteenused omamoodi erilised).
- ws-standardid (valmisarendatud sõnumikomplektid tüüpilisteks stsenaariumiteks nagu nt sisselogimine, et neid ise leiutama ei peaks)
- sisseehitatud veahaldus ja automaatika (teatud tasuliste toodete kasutamisel)

RESTil ei ole nii palju valmisfunktionaalsust, ent võrreldes SOAPIga on tal järgmised eelised:

- paindlik ja lihtne kasutada (kui mõni RESTi tingimustest ei meeldi, ei pea seda kasutama, kuigi siis öeldakse, et sinu REST teenus ei ole RESTful)
- veebiteenusega suhtlemiseks ei ole vaja lisatarkvara, sest HTTP pärngute tegemise võimalus ja JSON tugi on pea kõigis keeltes sees olemas.
- väiksem õppimiskõver (põhimõtteliselt töötab REST API server nagu tavoline veebiserver, väljastades HTML asemel JSONit)
- parem jõudlus / optimaalne võrguliikluse kasutus (SOAP kasutab kõikide sõnumite jaoks rangelt XML-i, mis on äärmiselt jutukas formaat ja kulutab info edastamiseks väga palju baite, RESTiga kasutatakse peamiselt JSON formaati, mis on palju kompaktsem ja lihtsam lugeda. Lisaks on RESTi puhul käsk HTTP meetod ja parameetrid URL-is, SOAPIl on aga alati POST meetod ja sama URL ning käsk ja parameetrid on päringu kehas XML elementide sees)

- REST on disainifilosofiliselt lähemal teistele veebitehnoloogiatele (kasutab HTTP enda meetodeid käskudena ja HTTP URL-i URI-na)

Siiski põhiline, mis eristab SOAPI RESTist on keerukus: REST on kergesti mõistetav ja rakendatav.

## Kuidas erineb URI RESTis ja SOAPIs?

RESTful API puhul on serveri andmebaasis paiknevaid andmed tehtud API kaudu kättesaadavaks nn **ressursside** või **kollektsooniidenä**. Igale kollektsoonile ehk ressursile on eraldatud omaette URL. Tehniliselt nad pole küll mitte URL-id, vaid URI-d (*unified resource identifier*). Mõte on selles, et nad eristavad eri tüüp andmeressursse serveris. Näiteks raamatupidamistarkvara REST API-I võib olla olemas URI-d /invoices, /accounts, /payments, /orders. Need peavad olema **nimisõnad** ja **mitmuses**. Näiteks ei sobi /car ega /getCar, vaid /cars. Sisselogimiseks on kollektsoon /sessions, kuhu elemendi (sessiooni) lisamisel on kasutaja sisse logitud. SOAPIs sõltub URI aga sellest, millist bindingut kasutatakse. Binding seob millegi millegagi. Antud kontekstis on mõeldud seda, et SOAPI meetodid saab siduda HTTP meetoditega, kui kasutada selleks spetsiaalselt väljatöötatud SOAP-i laiendust nimega SOAP HTTP Binding. Näiteks /accounting-web service

## Kuidas erineb käsu (ehk tegevuse, mida soovitakse teha) edastamine?

Kui SOAPIs kasutada andmeedastusviisiks HTTP protokolli, on valida kahe mustri vahel: [SOAP Request-Response message exchange pattern](#) ja [SOAP Response message exchange pattern](#). Esimese puhul saadab klient mistahes päringu alati POST päringuna ja andmed nagu mis funktsioon serveris käivitada ning milliste argumentidega, on kodeeritud POST päringu kehana saadetavas XML dokumendis; teine töötab rohkem nagu REST: need päringud, mis andmeid küsivad ja neid ei muuda, saadetakse GET päringuna ja vajalikud andmed edastatakse päringu rajana (/func1?arg1=foo&arg2=bar), kuid seda režiimi rakendatakse praktikas väga harva.

REST-i puhul edastatakse aga käsk kasutades HTTP meetodit, mis vastab kõige rohkem päringu olemusele:

- GET: Kollektsooni elementide loendi või ühe elemendi andmete hankimine.
- PUT: Elemendi andmete uuendamine (välja vahetamine uue komplekti andmetega).
- POST: Andmete saatmine töötluseks. Nt kollektsooni uue elemendi loomiseks või olemasoleva muutmiseks.
- PATCH: Olemasoleva elemendi üksikute andmete muutmiseks.
- DELETE: Andmete eemaldamine kollektsoonist.

Näiteks arve kustutamine tuleb kindlasti teha DELETE päringuga (nt `DELETE /invoices/42`), mitte POST päringuga (nt `POST /invoices?id=42&action=delete` või midagi sellist), sest DELETE on otseselt kustutamiseks, POST aga üldiselt andmete serverisse saatmine töötluseks ja alati tuleks eelistada seda meetodit. Kui on kahtlus, millist

HTTP meetodit peaks mingil momendil kasutama, tuleks pöörduda HTTP spetsifikatsiooni poole ja lugeda meetodite selgitusi. HTTP spetsifikatsioon kannab tähistust RFC 7231 ning meetodite kirjeldus on toodud selle spetsifikatsiooni sektsoonis 4.3. (otselink: <https://datatracker.ietf.org/doc/html/rfc7231#section-4.3>)

REST-is teeb klient HTTP päringuid vastava kollektsooni URI pihta. URI nimetatakse vahel ka lõpp-punktiks (*endpoint*). Päring on tavaline HTTP päring: algab nn päringu-reaga (Request-Line), mis sisaldab HTTP meetodit (kirjeldab ressursiga tehtavat toimingut) ja URI.

Näiteks GET meetod tähendab, et soovid lihtsalt andmeid lugeda, POST tähendab, et soovid luua uue ressursi, PATCH on uuenduste jaoks ja DELETE on andmete eemaldamiseks. Peale nende on veel mõned muud meetodid lisaks eelnevatele.



Joonis 2: HTTP päringu anatoomia

Pärast esimest rida on päised, mis sisaldavad metaandmeid taotluse kohta. Näiteks päisega Accept saab öelda serverile, et tahad andmeid mingis konkreetses formaadis saada ja päis Authorization sisaldab andmeid, mis tõendavad serverile, et sul on õigus seda päringut teha. Päistele järgneb keha ehk päringu sisu.

REST server võtab päringusõnumi vastu ja käivitab seejärel koodi, mis kontrollib vajadusel päisest juurdepääsuõigust, loeb siis andmebaasist soovitud andmed (või hoopis paneb nad sinna kui oli POST päring) ja seejärel vormindab andmetest vastussõnumi:



Joonis 3: HTTP päringu vastuse anatoomia

Sõnumi ülemine osa, staatus-rida, sisaldab staatuskoodi, mis ütleb kliendile, mis tema päringuga juhtus. Koodid 200-299 tähdavad, et kõik läks hästi. 400-499 koodid tähdavad, et sinu päringuga oli midagi valesti ja 500-599 tähdavad, et serveris oli mingi viga. Üks väga kasulik veebisait, kus on kõik staatuskoodid koos selgitustega ära toodud, on <https://httpstatuses.com>.

Pärast staatuskoodi on vastuse pääsed (*response headers*), mis sisaldavad teavet serveri kohta. Sellele järgneb vastuse keha (*response body*), mis sisaldab soovitud andmeid (*payload*) ja on API-de puhul tavaliselt vormindatud JSONis või XML formaadis.

REST arhitektuuri oluline osa on see, et see on olekuta (*stateless*), mis tähendab, et kaks osapoolt ei pea salvestama üksteise kohta mingit teavet ja iga päringu-vastuse tsükkel on sõltumatu kogu muust suhtlusest. Pole nii, et kõigepealt saadad ühe päringu, mis loob serveris mingi oleku ja alles siis saad edukalt teise saata (niiviisi käitub näiteks SMTP protokoll, millega e-maile saadetakse). See tagab, et veebirakendused oleksid korrektsed ja töökindlad.

## Harjutus 1.1: REST API demo käima saamine

1. Paigalda Node.js
2. Loo töölauale kaust `rest-api`
3. Käivita koodiredaktor (nt VS Code, WebStorm vms) ja ava see kaust projektina
4. Loo kausta fail `index.js` järgneva sisuga:

```
1  const express = require('express');
2  const cors = require('cors');
3  const app = express();
4
5  app.use(cors());           // Avoid CORS errors in browsers
6  app.use(express.json()) // Populate req.body
7
8  const widgets = [
9    { id: 1, name: "Cizzbor", price: 29.99 },
10   { id: 2, name: "Woooo", price: 26.99 },
11   { id: 3, name: "Craziinger", price: 59.99 },
12 ]
13
14 app.get('/widgets', (req :Request<P,ResBody,ReqBody,ReqQuery,Locals> , res :Response<ResBody,Locals> ) => {
15   res.send(widgets)
16 })
17
18 app.get('/widgets/:id', (req :Request<P,ResBody,ReqBody,ReqQuery,Locals> , res :Response<ResBody,Locals> ) => {
19   if (typeof widgets[req.params.id - 1] === 'undefined') {
20     return res.status( code: 404).send( body: { error: "Widget not found" })
21   }
22   res.send(widgets[req.params.id - 1])
23 })
24
25 app.post( path: '/widgets' , handlers: (req :Request<P,ResBody,ReqBody,ReqQuery,Locals> , res :Response<ResBody,Locals> ) => {
26   if (!req.body.name || !req.body.price) {
27     return res.status( code: 400).send( body: { error: 'One or all params are missing' })
28   }
29   let newWidget = {
30     id: widgets.length + 1,
31     price: req.body.price,
32     name: req.body.name
33   }
34   widgets.push(newWidget)
35   res.status( code: 201).location( url: 'localhost:8080/widgets/' + (widgets.length - 1)).send(
36     newWidget
37   )
38 })
39
40 app.delete( path: '/widgets/:id' , handlers: (req :Request<P,ResBody,ReqBody,ReqQuery,Locals> , res :Response<ResBody,Locals> ) => {
41   if (typeof widgets[req.params.id - 1] === 'undefined') {
42     return res.status( code: 404).send( body: { error: "Widget not found" })
43   }
44   widgets.splice( start: req.params.id - 1, deleteCount: 1)
45   res.status( code: 204).send( body: {error: "No content"})
46 })
47
48 app.listen( port: 8080, hostname: () => {
49   console.log(`API up at: http://localhost:8080`)
50 })
```

Kopeeri siit: <https://raw.githubusercontent.com/hajusrakendused/harjutus-1.1/main/index.js>

Seletus:

- Rida 1 laadib sisse Express.js raamistiku, mis on populaarne Node.js põhine raamistik API-de tegemiseks.

- Rida 2 laadib sisse cors paketi, mis võimaldab saata nn CORS päised päringu vastustega kaasa, mis lubavad API-t kasutada brauserist, juhul kui API server ja brauseris töötav kliendirakendus ei ole serveeritud ühest ja samast asukohast.
- Rida 3 initsialiseerib Express raamistiku (tekib app objekt)
- Rida 5 ütleb Expressile, et iga sissetuleva HTTP päringu puhul töödeldaks seda cors() funktsiooniga, mis lisab vastusesse CORS päised.
- Rida 6 ütleb Expressile, et iga sissetuleva HTTP päringu puhul töödeldaks seda express.json() funktsiooniga, mis analüüsib päringu keha ja kui seal on JSON, siis loeb JSONist parameetrid req.body objekti
- Read 8-12 defineerivad widgets nimelise massiivi, millel on 3 liiget, mis on objektid, millel on 3 atribuuti: id, name ja price
- Widget on [kohatäite nimetus geneerilisele tootele](#). Näiteks kui meil oleks filmide andmebaas, siis oleks widget asemel vaja kasutada sõna movie.
- Read 14-16 defineerivad lõpp-punkti GET /widgets päringle, mis saadab vastusena terve widgets massiivi (mis on defineeritud ridadel 8-12)
- Read 18-23 defineerivad lõpp-punkti GET /widgets/id päringle, mis saadab vastusena selle id-ga vastaneva vidina widgets massiivist (defineeritud ridadel 8-12), mis on lisatud /widgets/id asemele. Kui päringul on selline id, mida widgets massiivis ei ole defineeritud, saadetakse tagasi veateade "Widget not found" HTTP staatuskoodiga 404
- Read 25-38 defineerivad lõpp-punkti POST /widgets päringle, mis lisab kollektiooni uue widgeti
  - Read 26-28 kontrollivad, et päringul oleks kõik kohustuslikud parameetrid uue vidina lisamiseks
  - Read 29-33 defineerivad, millised on kohustuslikud parameetrid uue vidina lisamiseks. Uue vidina id saamiseks liidetakse massiivis elevate vidinate arvule juurde 1
  - Rida 34 defineerib uue vidina lisamist widgets massiivi
  - Read 35-38 defineerib vastuse eduka päringu puhul staatuse 201
- Read 40-46 defineerivad lõpp-punkti DELETE /widgets/id päringle, mis kustutab vastusena selle id-ga vastaneva vidina widgets massiivist (defineeritud ridadel 8-12), mis on lisatud /widgets/id asemele. Kui päringul on selline id, mida widgets massiivis ei ole defineeritud, saadetakse tagasi veateade "Widget not found" HTTP staatuskoodiga 404.
  - Real 44 kasutame funktsiooni splice, millega eemaldame vidina widget massiivist. Esimeseks funktsiooni parameetriks anname alguse ehk 0 koha massiivist, kust hakatakse vidinat eemaldama. Teiseks parameetriks anname arvu, mitu elementi eemaldatakse.
  - Real 45 anname õige tagastatava staatuskoodi: `204 No Content`
- Read 48-50 kutsutakse välja meetod listen(). Esimeseks argumendiks on port 8080, millel hakatakse kuulama pärnguid. Teiseks (valikuline) argumendiks funktsiooni, kus prinditakse konsooliile aadressi, millel on rakendus kätesaadav

5. Käivita koodiredaktoris terminal ja seal järgnevad käsud:

```
npm init -y
npm i express cors
node .
```

6. Paigalda `xh` (kiire tööriist HTTP pärngute saatmiseks), mille saad Githubist

7. Tee terminalis `xh`-ga GET püring vastu API-t, mis tagastab kõik vidiinad:

```
$ xh -v localhost:8080/widgets
GET /widgets HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Host: localhost:8080
User-Agent: xh/0.14.1

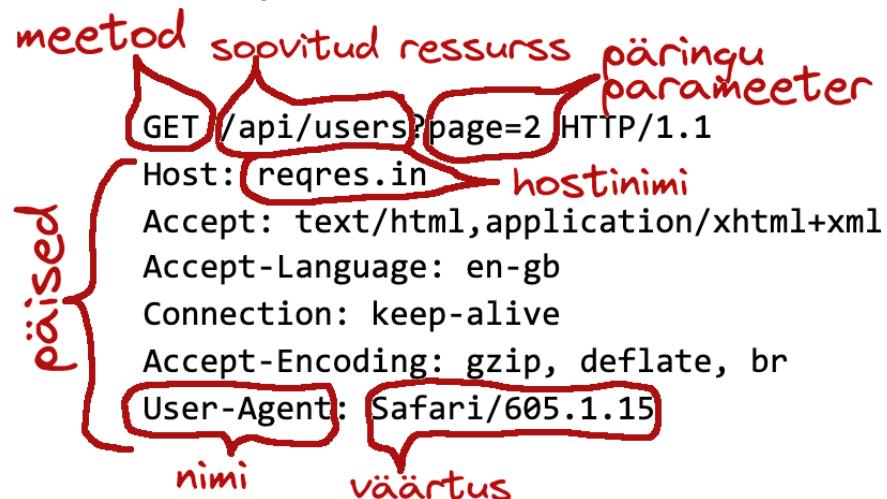
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 122
Content-Type: application/json; charset=utf-8
Date: Wed, 12 Jan 2022 13:05:00 GMT
Etag: W/"7a-w9EJN9ReyDAeKiz0FwwcdTjFIjQ"
Keep-Alive: timeout=5
X-Powered-By: Express

[
  {
    "id": 1,
    "name": "Cizzbor",
    "price": 29.99
  },
  {
    "id": 2,
    "name": "Woowo",
    "price": 26.99
  },
  {
    "id": 3,
    "name": "Crazlinger",
    "price": 59.99
  }
]
```

Seletus:

- `-v` tähendab, et näita ka kliendi poolt serverisse saadetavat püringut. Vaikimisi `xh` kuvab ainult püringu vastuse osa.
- `localhost:8080/widgets` on API lõpp-punkt, kuhu HTTP püring tehakse

- Siin on HTTP päringu anatoomia lahtiseletus:



- Tee terminalis `xh`-ga GET pärting vastu API-t, mis tagastab ühe konkreetse vidina:

```

$ xh -v localhost:8080/widgets/1
GET /widgets/1 HTTP/1.1
Accept: /*
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Host: localhost:8080
User-Agent: xh/0.14.1

HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 39
Content-Type: application/json; charset=utf-8
Date: Wed, 12 Jan 2022 13:05:00 GMT
Etag: W/"7a-w9EJN9ReyDAeKiz0FwwcdTjFIjQ"
Keep-Alive: timeout=5
X-Powered-By: Express

{
  "id": 1,
  "name": "Cizzbor",
  "price": 29.99
}
  
```

Seletus:

- `localhost:8080/widgets/1` on API lõpp-punkt, kuhu HTTP pärting tehakse
- `GET` pärtingut kasutatakse API-it andmete saamiseks

9. Tee terminalis `xh`'ga POST püring vastu API-t, mis lisab uue vidina:

```
$ xh -v localhost:8080/widgets name=Fozzockle price=39.99
POST /widgets HTTP/1.1
Accept: application/json, */*;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 36
Content-Type: application/json
Host: localhost:8080
User-Agent: xh/0.14.1

{
    "name": "Fozzockle",
    "price": "39.99"
}

HTTP/1.1 204 No Content
Access-Control-Allow-Origin: *
Connection: keep-alive
Date: Thu, 03 Feb 2022 14:50:39 GMT
Etag: W/"16-5pd0zcCmKVgY+xEFyBGnRC6U2N4"
Keep-Alive: timeout=5
X-Powered-By: Express

{
    "id": 4,
    "price": "39.99",
    "name": "Fozzockle"
}
```

Seletus:

- `localhost:8080/widgets` on API lõpp-punkt, kuhu HTTP pääring tehakse
- `name=...` ja `price=...` on kohustuslikud parameetrid uue objekti lisamiseks andmekollektsiooni `widgets`
- `id` genereeritakse ja lisatakse widgetile serveri poolt

10. Tee terminalis `xh`'ga POST pääring vastu API-t, mis kustutab ühe vidina:

```
$ xh -v DELETE localhost:8080/widgets/2
DELETE /widgets/2 HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Host: localhost:8080
User-Agent: xh/0.15.0

HTTP/1.1 204 No Content
Access-Control-Allow-Origin: *
Connection: keep-alive
Date: Thu, 03 Feb 2022 14:50:39 GMT
Etag: W/"16-5pdOzcCmKVgY+xEFyBGnRC6U2N4"
Keep-Alive: timeout=5
X-Powered-By: Express
```

Seletus:

- `localhost:8080/widgets/2` on API lõpp-punkt, kuhu `DELETE` HTTP pääring tehakse
- widget, mille `id` on 2, kustutatakse massiivist

## 1.2 Andmete lugemine ja töötlemine JSON andmevahetusformaadist

### Mis on JSON?

JSON on andmevahetusformaat, mida on lihtne analüüsida ja genereerida. JSON on objekti andmete kirjeldamiseks JavaScript'is kasutatava süntaksi laiendus. Ometi ei ole see piiratud JavaScript'iga kasutamisega. See on tekstivorming, mis kasutab andmete kaasaskantavaks esitamiseks objekti- ja massiivstruktuure. Kõik kaasaegsed programmeerimiskeeled toetavad JSON andmestruktuure, mis teeb JSONi keelest sõltumatuks. JSON kasutus on ülimalt populaarne REST API-de puhul.

## JSON struktuur

JSON-objekti struktuur on järgmine:

- looksulgudes {} hoitakse objekte, mis on komadega eraldatud andmed välti:väärustus formaadis;
- võtmed on alati ümbratsetud jutumärkidega (erinevalt JavaScriptist);
- võtmete kaudu saab objektist konkreetseid andmeid küsida;
- nurksulgudes [] hoitakse massiive, mis võivad sisaldada 0 kuni lõpmatus (kuni mälu jätkub) arvul elemente, mis võivad olla stringid, objektid, teised massiivid või muud sorti andmed, mida JSON toetab;
- kui väärus on string, on ta ümbratsetud jutumärkidega.

```
[{"Widget": "Modal"}, {"Widget": "Cursor changer", "Price": 20.00}]
```

Näide JSON struktuurist (kompaktsel kujul)

```
{
  "Microsoft": {"Widget": "Cursor changer", "Price": 20.00}
}
```

Ühe objekti sees on teine objekt.

```
[
  "Microsoft": {"Widget": "Cursor changer", "Price": 20.00},
  "SomeGuyOnReddit": {"Widget": "Darkmode", "Price": 8.19}
]
```

Mitu objekti, mille väärusteks on objektid.

```
{
  "Microsoft": {"Widget": "Cursor changer", "Price": 20.00},
  "Microhard": [
    {"Widget": "Darkmode", "Price": 8.19},
    {"Widget": "Background color randomizer", "Price": 8.19}
  ]
}
```

Objektis *Microhard* pesitseb mitu objekti massiivi sees

Objektid ja massiivid on väärtsused, mis võivad sisaldada teisi väärtsusi, seega on JSON-andmetega võimalik piiramatu pesastamine (*nesting*). See võimaldab JSON'is kirjeldada enamikku andmetüüpe, alates tabelitest kuni veelgi keerulisemate andmetüüpide.

## JSON andmetüübidi

Olles nüüd tutvunud JSON-i struktuuriga, oled juba ka tuttav mõndade JSON-i andmetüüpidega. Siin on täielik loetelu neist:

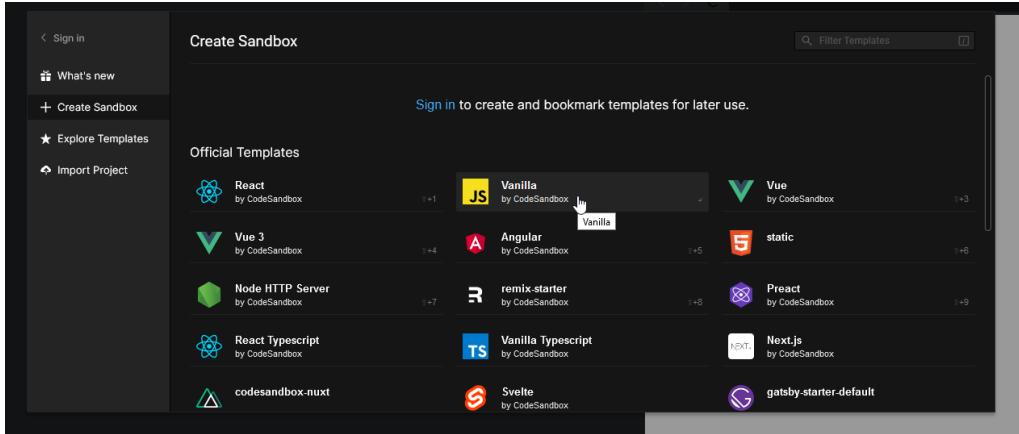
- string - sõnaline tekst, mis on ümbritsetud jutumärkidesse.
- number - positiivsed või negatiivsed täisarvud või ujukomaarvud.
- objekt - võtme ja väärtsuse paar, mis on ümbritsetud looksulgude sisse.
- array - massiv, kuhu saab pesitseda mitmeid JSON-objekti kogumeid.
- boolean - Väärtsus true või false ilma jutumärkideta.
- null - Näitab andmete puudumist võtmeväärtsuspaari puhul, mida esitatakse kui "null" ilma jutumärkideta.

```
{  
  "myString": "foo",  
  "myNumber": 10,  
  "myObject": {  
    "myString": "bar",  
    "myNumber": 1,  
    "myObject": {  
      "myString": "baz",  
      "myNumber": 0.1  
    }  
  },  
  "myRandomContentArray": [  
    "foo",  
    10,  
    {  
      "myString": "foo",  
      "myNullValue": null,  
      "myTruthyValue": true,  
      "myFalseyValue": false  
    }  
  ],  
  "myArrayOfObjects": [  
    {  
      "id": 1,  
      "name": "PewDiePie",  
      "link": "https://www.youtube.com/user/PewDiePie"  
    },  
    {  
      "id": 2,  
      "name": "MrBeast",  
      "link": "https://www.youtube.com/user/MrBeast6000"  
    },  
    {  
      "id": 1,  
      "name": "5-Minute Crafts",  
      "link": "https://www.youtube.com/channel/UC295-Dw_tDNTZXFeAPA6Aw"  
    }  
  ]  
}
```

Näide kõikidest võimalikest andmetüüpidest.

## Harjutus: Loo [Codesandbox](#)-is HTML leht, mis kuvab auto andmeid

1. Ava veeblehitsejas Code Sandbox sait
2. Vali kollase taustaga Vanilla



3. Kirjuta pildil olev kood:

A screenshot of the CodeSandbox development environment. On the left, the 'index.js' file is open in a code editor with the following content:

```
1 import "./styles.css";
2
3 const myjson = [
4   {
5     Car: {
6       Color: "Rose red",
7       "Tinted windows": true
8     }
9   }
10 ];
11
12 document.getElementById("app").innerHTML =
13 <div id="json">
14   <h1> Car properties </h1>
15   <p>Color: ${myjson[0].Car.Color}</p>
16   <p>Tinted windows: ${myjson[0].Car["Tinted windows"]}</p>
17 </div>
18 `;
19 
```

On the right, the 'Browser' tab shows the rendered output: 'Car properties' with 'Color: Rose red' and 'Tinted windows: true'. The URL in the browser bar is https://dsmuk.csb.app/.

4. Lisa muutujasse myjson ja vastavalt ka <div> elementi juurde puuduvad andmed alljärgnevast tabelist:

Property	Car 0	Car1
Color	Rose red	Navy blue
Tinted windows	FALSE	TRUE
Wheels	4	4
Roof cargo	null	Thule
Entertainment	FM Radio MP3, MP4 and MKV player harman/kardon speakers	FM Radio Apple CarPlay/Android Auto Bowers & Wilkins Premium Sound speakers
Accessories	satnav, cruise control	self drive system, luggage cover

5. Lõpptulemus võiks välja näha selline:

The screenshot shows a browser window with the address bar displaying 'https://bul8t.csb.app/'. The main content area has a dark header with 'Browser' and 'Tests' tabs. Below the header, there are navigation buttons ('< >'), a refresh button, and a URL input field. The main content is titled 'Car properties' in large bold letters. Below the title, there is a list of car features with their values: 'Color: Rose red', 'Tinted windows: true', 'Wheels 4', 'Roof cargo: null', 'Audiosystem: Radio,Appleplay', and 'Accessories: navigation system, self drive system, luggage cover, roof rack'.

## 1.3 Klientrakenduse ehitamine REST teenusele

Vaata eraldi faili "Õppaineid läbiv projekt"

<https://docs.google.com/document/d/18HFxAd1PPtds7mEnMXZCvubnE7q9tjeM9JB4-4vWG1Y/edit#>

## 1.4 Veobiteenuse optimeerimine ja andmete puhverdamine ning dubleerimine;

Andmete puhverdamine (*data caching*) on protsess, mis salvestab andmeid või faili ajutisse salvestuskohta ehk vahemällu, et neile saaks kiiremini ligi pääsedada. See salvestab andmeid tarkvararakenduste, serverite ja veebibrauserite jaoks, mis tagab, et kasutajad ei pea veeblehe või rakenduse laadimise kiirendamiseks iga kord teavet alla laadima.

Andmete dubleerimine (*data replication*) on protsess, mille käigus tehakse andmetest koopiaid ja hoitakse neid varundamise eesmärgil eri kohtades: samas süsteemis, füüsilises/virtuaalses serveris või pilvepõhiselt.

Andmete dubleerimist kasutatakse, et:

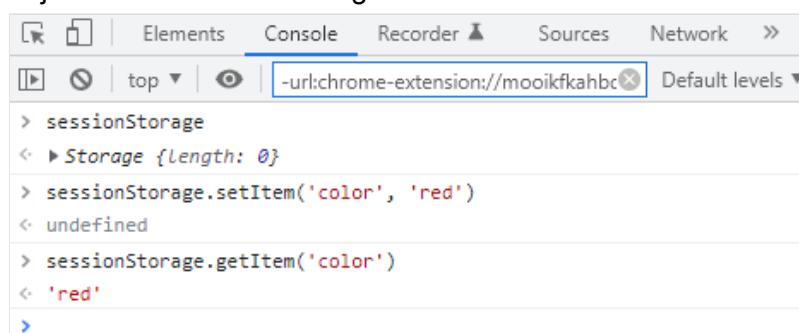
- parandada andmete kättesaadavust
- suurendada andmetele juurdepääsu kiirust
- parandada serveri jõudlust
- andmeid taastada

## localStorage and sessionStorage

localStorage ja sessionStorage võimaldavad salvestada vältme-väärtuse paare lokaalselt. SessionStorage'i puhul andmed säilivad lehe uuendamisel ja localStorage'i puhul kuni kasutaja kustutab brauseri vahemälu käsitsi või kuni veebirakendus kustutab andmed.

### Harjutus.sessionStorage

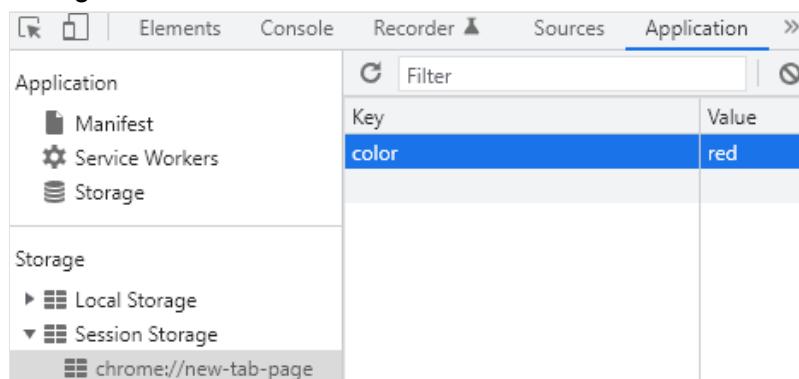
1. Ava brauser ja mine `chrome://newtab` lehele.
2. Ava konsool ning kirjuta konsooliile `sessionStorage` ning vajuta ENTER. Näed, et Session Storage on tühi.
3. Tee uus käsk `sessionStorage.setItem('color', 'enda lemmikvärv')`. Selle käsga lisati Session Storage'isse uued andmed.
4. Kirjuta uus käsk `sessionStorage.getItem('color')` ning näed, et väljastatakse sessionStorage'ist sinu lemmikvärv.



The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. The command history is visible:

```
> sessionStorage
<- > Storage {length: 0}
> sessionStorage.setItem('color', 'red')
<- undefined
> sessionStorage.getItem('color')
<- 'red'
```

5. Et näha, mis on Session Storage'isse salvestatud, mine Application>Session Storage>vastav domeen



The screenshot shows the Chrome DevTools interface with the 'Application' tab selected. The left sidebar shows 'Storage' expanded, with 'Session Storage' selected. The main area displays a table of stored items:

Key	Value
color	red

6. Nüüd ava veel üks vahekaart ning mine samale lehele (`chrome://newtab`). Aва uesti Application>Session Storage ning näed, et see on tühi.

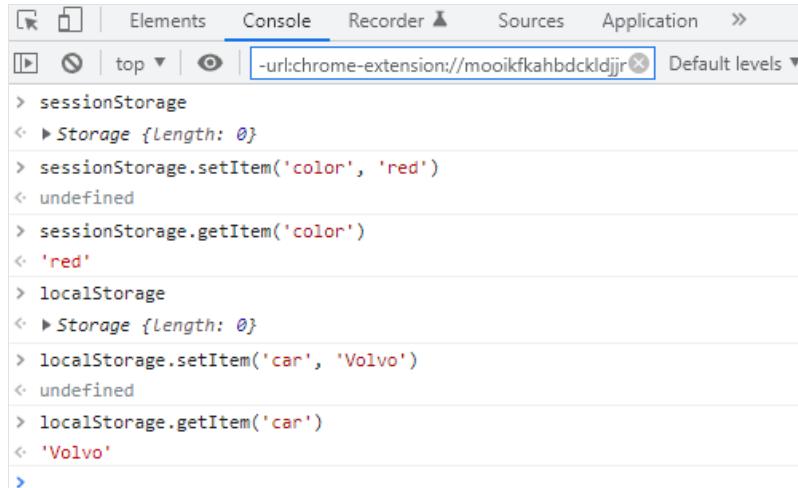
Seletus:

- Session Storage'isse salvestatakse andmed ainult konkreetse seansi jaoks. See, kus tegid `sessionStorage.setItem()` jne käske on üks seanss, ning teine vahekaart, mille 6.punkti juures avasid, oli teine seanss.

### Harjutus.localStorage

1. Mine sellele vahekaardile tagasi, kus viimati sessionStorage käske sooritasid.

2. Ava uuesti konsooli ning kirjuta `localStorage`. Näed, et see on tühi.
3. Tee uus käsk `localStorage.setItem('car', 'enda lemmikauto')`. Selle käsga lisati Local Storage'isse uued andmed.
4. Soorita käsk `localStorage.getItem(car)` ning näed, et localStoragesse väljastatakse sinu lemmikauto.

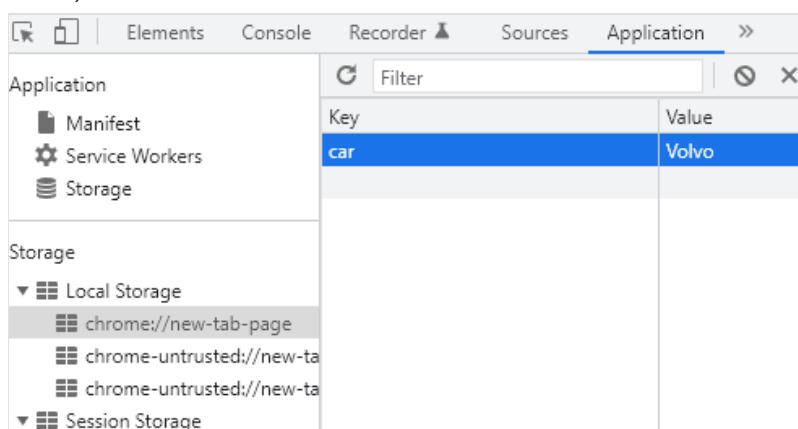


```

Elements  Console  Recorder  Sources  Application  >
top  -url:chrome-extension://mooikfkahbdckldjjr  Default levels
> sessionStorage
< ▶ Storage {length: 0}
> sessionStorage.setItem('color', 'red')
< undefined
> sessionStorage.getItem('color')
< 'red'
> localStorage
< ▶ Storage {length: 0}
> localStorage.setItem('car', 'Volvo')
< undefined
> localStorage.getItem('car')
< 'Volvo'
>

```

5. Nüüd tee uuesti uus vaheaken ning vaata Application>Local Storage alla. Näed, et see on sinna salvestatud.



Key	Value
car	Volvo

Storage

- Local Storage
  - chrome://new-tab-page
  - chrome-untrusted://new-ta
  - chrome-untrusted://new-ta
- Session Storage

Seletus:

- Local Storage'sis säilitatakse andmed kõigis sessioonides antud konkreetses domeenis.

## Küpsised

Kui klient saadab serverile HTTP päringu, luuakse serveri [TCP](#) pordiga koriks sessioon, mis on kahesuunaline sidekanal, kuid HTTP on tehtud nii, et esmalt saadab klient päringu ja siis server vastab sellele ning katkestab kohe ühenduse ära. Kui klient soovib pärast seda teha uue päringu, peab ta looma uue TCP ühenduse aga serveri poolt on näha ainult klienti IP aadress ja port. Aga sama IP aadressi taga võib olla mitu klienti ja klienti võib oma IP aadressi vahetada (nt WiFi ühenduselt 4G peale üle minnes või vastupidi). Seega kui klient uue päringu teeb, pole serveril aimugi, kas see on see sama klient või mitte.

Küpsised (*cookies*) on nime-väärtuse paarid, mida veebiserver saab kliendile saata ning mida klient võib serverile järgneva(te) päringu(te)ga tagasi saata, kuni küpsis aegub.

## Harjutus

1. Saada `xh` käsurearakendusega Twitteri veebiserverile päring ning vaata, millised küpsised Twitteri veebiserver tagasi annab.
2. Uuri netist `set-cookie` päise anatoomiat (MDN Web Docs saidil on päris hea kirjeldus) ning analüüs, millist infot, kellele ja kui kauaks Twitter sinu brauserisse üritas sokutada.

```
hennotaht@Hennos-MacBook-Pro ~> xh -h https://www.twitter.com
set-cookie: guest_id_marketing=v1%3A164210783548879484; Max-Age=63072000; Expires=Sat, 13 Jan 2024 21:03:55 GMT; Path=/; Domain=.twitter.com; Secure; SameSite=None
set-cookie: guest_id_ads=v1%3A164210783548879484; Max-Age=63072000; Expires=Sat, 13 Jan 2024 21:03:55 GMT; Path=/; Domain=.twitter.com; Secure; SameSite=None
set-cookie: personalization_id="v1_rYHrEMbytaTCV+jd60yepA=="; Max-Age=63072000; Expires=Sat, 13 Jan 2024 21:03:55 GMT; Path=/; Domain=.twitter.com; Secure; SameSite=None
set-cookie: guest_id=v1%3A164210783548879484; Max-Age=63072000; Expires=Sat, 13 Jan 2024 21:03:55 GMT; Path=/; Domain=.twitter.com; Secure; SameSite=None
```

Küpsised võimaldavad serveril eristada kliente üksteisest ja võimaldavad sisselogimisfunksionaalsust, saidi eelistuste salvestamise funksionaalsust, ostukorvi jms, mis eeldab, et server tunneb kliendi ära.

## Kuidas localStorage töötab?

Kui küpsised on andmed, mis liiguvad päringute päistes, siis localStorage on brauseri funksionaalsus, mis võimaldab kliendi poolsel Javascriptil (see JS, mis jookseb brauseris, mitte serveris, kuid mille server on üldjuhul kliendile käivitamiseks saatnud) kasutada kohalikku kövaketast, et vajalikke andmeid seal hoida. Näiteks puhverdamise eesmärgil.

Oluline on see, et kui localStorage'isse midagi panna, siis teiste domeenide serveritest (või isegi samal domeenil, aga teise porti otsas ja isegi http vs https on erinevad serverid localStorage'i jaoks) ei saa seda lugeda. Piltlikult öeldes näiteks Facebook ei saa lugeda, mida Gmaili veebirakendus brauseri localStorage'isse pani.

LocalStorage objektil on viis meetodit:

- `setItem()`: Võtme ja väärtuse lisamiseks localStorage'isse
- `getItem()`: localStorage'ist objekti väärtuste saamiseks
- `removeItem()`: Elemendi objekti võtme järgi localStorage'ist eemaldamiseks
- `clear()`: kogu localStorage'i tühhjendamiseks
- `key()`: Antud number localStorage'i võtme leidmiseks

## Näide

Allolevalt toome näite, kuidas localStorage'it saab andmete puhverdamiseks kasutada:

```

1  widgetsCache = typeof localStorage.getItem( key: 'widgetsCache' ) !== 'undefined'
2      ? JSON.parse(localStorage.getItem( key: 'widgetsCache' ))
3      : await (await fetch( input: 'http://localhost:8080/widgets')).json();
4
5  localStorage.setItem('widgetsCache', JSON.stringify(widgetsCache))

```

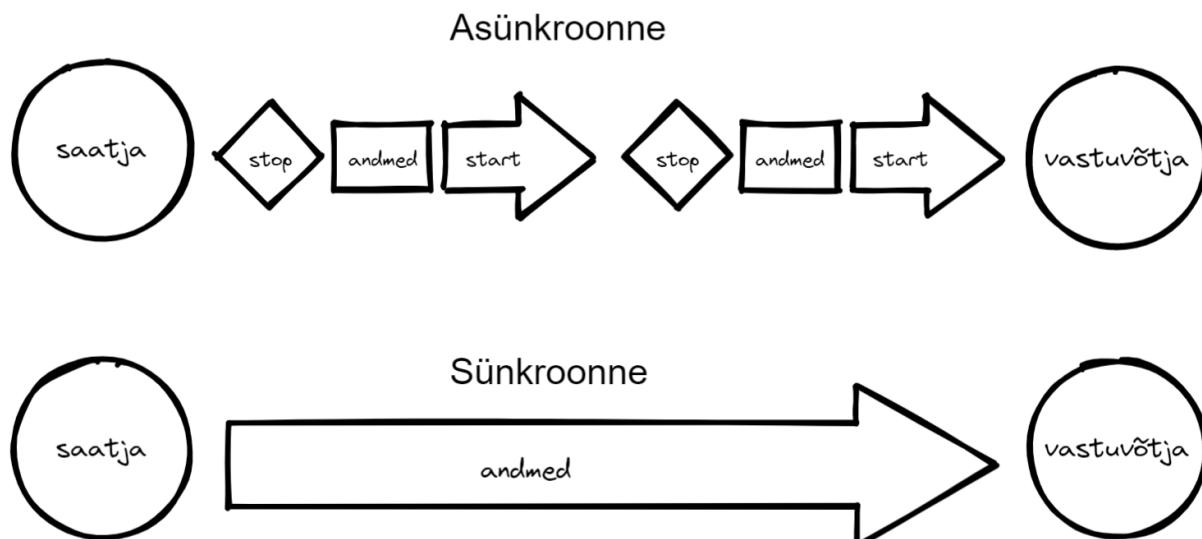
Selgitus:

- **Real 1** kontrollitakse, kas localStorage'is on miskit võtme widgetsCache all
- **Rida 2:** kui localStorage'is oli selle võtme all miskit, antakse see widgetsCache'ile väärtsuseks. Enne väärvtustamist tehakse eeldus, et tegemist on JSON formaadis tekstiga ja teisendatakse see JSON-ist Javascript objektiks.
- **Rida 3:** kui localStorage'is ei olnud selle võtme all midagi, antakse widgetsCache'ile väärtsuseks võrgupäringu tulemusel saadud JSONist parsitud objekt (või mis iganes andmetüüp, mis parsimisel saadi).
- **Real 5** salvestatakse widgetsCache'i hetkeväärtsus localStorage'isse, et seda järgmisel korral ilma võrgupäringut tegemata saaks kasutada.

## 1.6 Asünkroonne ja sünkroonne andmete vahetus veebiteenusega;

Asünkroonsete operatsioonide puhul saab minna teise ülesande juurde enne kui eelmine ülesanne lõpeb. Nii saab asünkroonse programmeerimise puhul tegeleda mitme päringuga samaaegselt, lõpetades seega rohkem ülesandeid palju lühema ajaga.

Sünkroonsete puhul täidetakse ülesandeid ükshaaval ja alles siis kui üks neist on lõpetatud, alustatakse järgmisi. Teisisõnu, tuleb oodata ühe ülesande lõpetamist, et liikuda järgmise juurde.



## AJAX

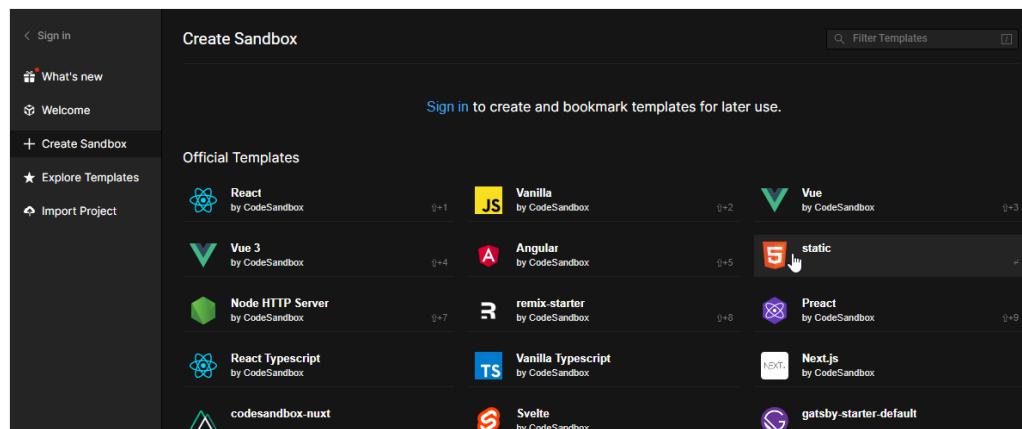
AJAX (Asynchronous JavaScript And XML) on meetod andmete vahetamiseks serveriga ja veebilehe osade uuendamiseks, ilma et peaks kogu lehekülge uuesti laadima. AJAX ei ole programmeerimiskeel.

AJAX kasutab kombinatsiooni:

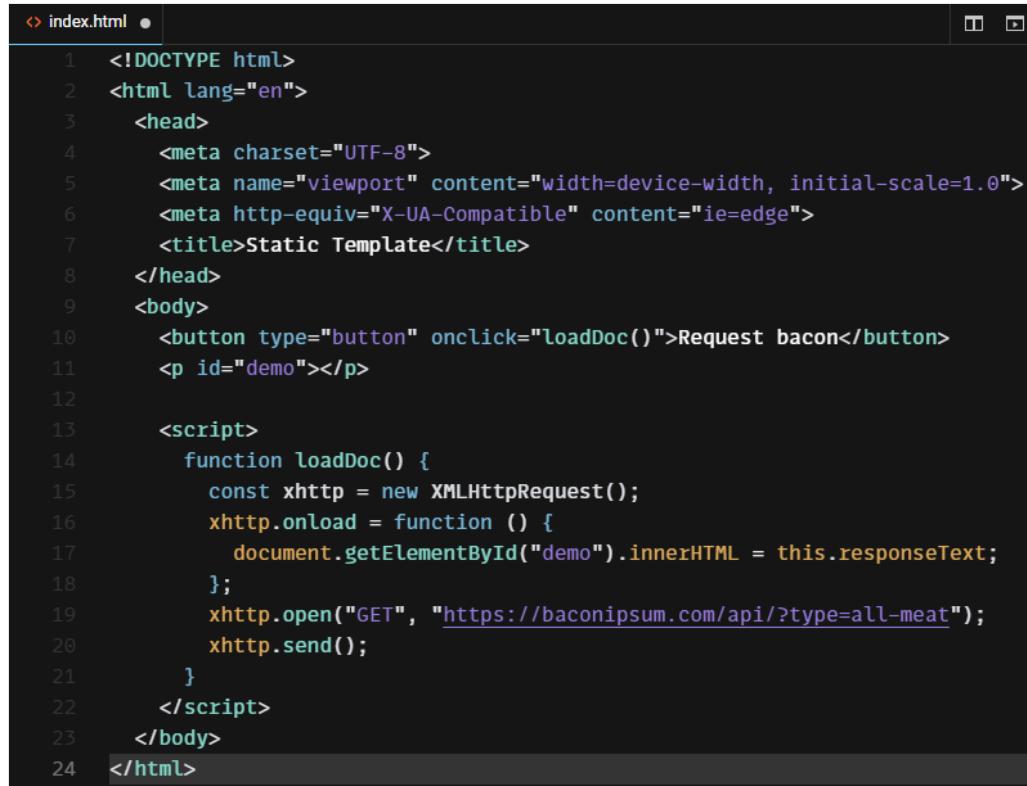
- Brauseri sisseehitatud XMLHttpRequest objekti (andmete taotlemiseks veebiserverist)
- JavaScript ja HTML DOM-i (andmete kuvamiseks või kasutamiseks)

## Harjutus

1. Ava veeblehitsejas Code Sandbox sait
2. Vali *Official Templates* alt *static*



3. Kirjuta pildil olev kood index.html faili. Alustuseks kasuta HTML trafaretti (hüümärk ja tab klahv).



```
index.html
1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <meta charset="UTF-8">
5          <meta name="viewport" content="width=device-width, initial-scale=1.0">
6          <meta http-equiv="X-UA-Compatible" content="ie=edge">
7          <title>Static Template</title>
8      </head>
9      <body>
10         <button type="button" onclick="loadDoc()">Request bacon</button>
11         <p id="demo"></p>
12
13         <script>
14             function loadDoc() {
15                 const xhttp = new XMLHttpRequest();
16                 xhttp.onload = function () {
17                     document.getElementById("demo").innerHTML = this.responseText;
18                 };
19                 xhttp.open("GET", "https://baconipsum.com/api/?type=all-meat");
20                 xhttp.send();
21             }
22         </script>
23     </body>
24 </html>
```

4. Salvesta fail CTRL + S
5. Lõpptulemuseks peaksid saama andmeid peekoni kohta

## 1.7 Mõne avaliku teenusepakkuja veebiteenusega liidestumine

Paljud suuremad teenusepakkujad pakuvad võimalust kasutada nende andmeid läbi veebiteenuse ehk veebipõhise (st HTTP-d kasutava) API.

On olemas saite, mis agregeerivad sellised avalikud API-d kokku. Näiteks need saidid:

- [Free APIs](#)
- [Dev Resources](#)
- [Public APIs Site](#)
- [Apihouse](#)
- [Collective APIs](#)

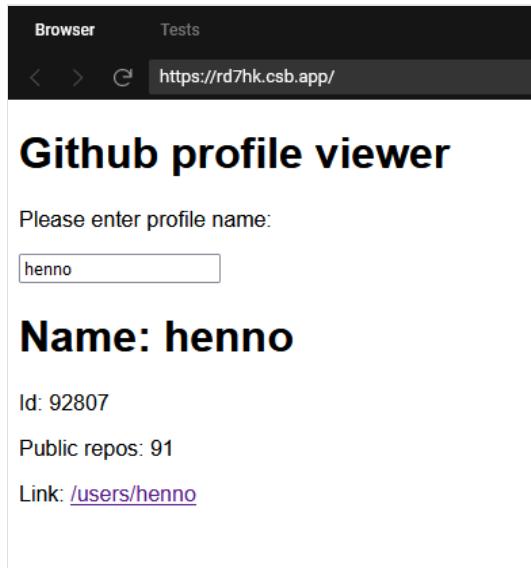
Vaata neid saite, et tutvuda, millist avalikku informatsiooni on võimalik enda rakenduses kasutamiseks API teel saada.

## Harjutus: Github andmete kasutamine API kaudu

Käesolevas harjutuses õpid võtma andmeid Githubi avalikust API-st. Proovime pärida GitHubi kasutaja profiili andmeid.

Link dokumentatsioonile: [Github API](#)

Harjutuse lõpuks valmib selline rakendus:



Sammud:

1. Valmista endale uus *Vanilla* keskkond [Code Sandbox](#)'is.

2. Kasuta järgmist malli *index.js* failis:

### *index.js*

```
let givenProfile = "";
let profileName = "";
let profileId = "";
let profileLink = "";
let profileRepos = "";

function renderPage() {
  document.getElementById("app").innerHTML =
    <div>
      <h1>Github profile viewer</h1>
      <p>Please enter profile name: </p>
      <input />
      <div class="content">
        <h1 id="name">Name: ${profileName}</h1>
        <p id="id">Id: ${profileId}</p>
        <p id="repos">Public repos: ${profileRepos}</p>
        <p id="profileurl">Link: ${profileLink}</p>
        <a href="#">${profileName}</a>
        </div>
      </div>
    `;
}

renderPage();
```

3. Tee JavaScript *fetch* püring. Kasuta esimest näidet *developer.mozilla* dokumentatsioonis.
4. *Fetch*'i vastu Github kasutaja API-t. Uuri teist näidet profiili saamiseks [siit](#).
5. Uuri konsoolist (*console.log*) *fetch*'itud vastust.

```
▶ {login: "defunkt", id: 2, node_id: "MDQ6VXNlcjI=", avatar_url: "https://avatars.githubusercontent.com/u/2?v=4", gravatar_id: "..."}
```

6. Leia vastusest: login, id, html\_url ja public\_repos võtmed.
7. Lisa *input* elemendile funktsionaalsus (*index.js* lõppu).

```
const input = document.querySelector("input");
input.addEventListener("change", updateValue);

function updateValue(e) {
    givenProfile = e.target.value;
    fetchProfile();
}
```

8. Tee `fetch`'ist funktsioon. Lisa muutujad, pannes pärast punkti API-lt tahetud võtmed ja anna `fetch` URL-ile õige muutuja. Mõtle, mis võti võiks millise muutujaga seotud olla.

```
let fetchProfile = async () => {
    let fetchedData;

    await fetch(`https://api.github.com/users/${givenProfile}`)
        .then((response) => response.json())
        .then((data) => (fetchedData = data));

    profileName = fetchedData.name;
    profileId = fetchedData.id;
    profileLink = fetchedData.html_url;
    profileRepos = fetchedData.repos_url;

    renderPage();
};


```

9. Kui oled teinud õigesti, siis `input` kasti sisestades kellegi Github profiili nime, saad selle profiili kohta infot lehele.

# ÕV2: Loob veebiteenuseid, kasutades sünkroonseid ja asünkroonseid andmete ülekandmise võimalusi, valides neist sobivaima lähteülesande lahendamiseks

**Soovituslik tölgendus sellele õpiväljundile:** loob ja dokumenteerib veebiteenuseid, sh. masinloetava spetsifikatsiooniga;

kirjeldab hajusrakenduste olemust ja kasutusvaldkondi;

- loob ja paigaldab veebiteenuseid, kasutades nii olekuga kui olekuta tehnoloogiaid (nt SOAP, WEBAPI; REST);
- kasutab süsteemidevahelist sünkroonset ja asünkroonset andmete ülekandmist;
- selgitab, milleks on vajalik andmete puhverdamine (caching) ja liiasusega hoiustamine, lähtudes käideldavuse tagamise põhimõttest;

kasutab enamlevinud raamistikke ja rakendusservereid<sup>[AV1]</sup> veebiteenuste realiseerimiseks.

## 2. HAJUSSÜSTEEMIDE OLEMUS JA ERINEVAD TÜÜBID

### 2.1 Hajusa arhitektuuriga lahenduse eelised ning kasutusvaldkonnad

Hajusrakendus on rakendus, mis on jagatud väiksemateks rakendusteks, mis omavahel võrgu kaudu suhtlevad.

**Hajutatuse seisukohast eristatakse järgnevaid rakenduse arhitektuurimudeleid:**

- monoliitne arhitektuur (midagi ei ole jaotatud),
- klient-server arhitektuur (üks server, mitu tarbijat ehk klienti),
- mikroteenuste arhitektuur (server koosneb mitmest mikroserverist, mille poole kliendid otse või läbi lüüsi pöörduvad ja mis võivad ka üksteisega andmeid vahetada).

### Rakenduse osadeks hajutamise eesmärgid:

- kasutada mitme riistvaraseadme ressursse ära,
- minimeerida terve rakenduse tööseisakut,
- võimaldada ühe suure arendusmeeskonna asemel, kus keegi ei vastuta millegi eest, tekitada selgepiirilised vastutusalad, kus iga meeskond on vastutav oma tükki eest.
- erinevaid rakenduse osasid on võimalik kirjutada erinevates programmeerimiskeeltes.

#### Kontrollküsimused

1. Millised on 3 hajusrakenduse arhitektuurimudelit?
2. Millised võimalused annab hajus arhitektuur meeskondade töökorraldusele?
3. Kuidas mõjutab hajus arhitektuur rakenduse töökindlust?
4. Millise probleemi lahendab hajus arhitektuur?

## 2.2 Andmete lugemine ja töötlemine XMList

### Mis on XML?

XML tähendab eXtensible Markup Language ja on HTML-i sarnane märgistuskeel. XML on loodud andmete salvestamiseks ja edastamiseks. XML-keelel ei ole etteantud silte. XML-märgendid määravad andmete struktuuri ja tähenduse – mis andmed need on.

XML-failid on kodeeritud lihtkirjas, nii et neid saab avada mis tahes tekstiredaktoris ja neid saab selgelt lugeda. Peaaegu iga brauser saab XML-faili avada. Chrome'is avage lihtsalt uus vahekaart ja lohistage XML-fail üle. Teine võimalus on klõpsata XML-failil paremal hiireklõpsuga ja valige "Open with" ning seejärel klõpsake "Chrome". Peale seda avaneb fail uues vahekaardis.

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
    <book>
        <name>JavaScript Garden</name>
        <author>Ivo Wetzel</author>
    </book>
    <book>
        <name>You Don't Know JS: Up & Going 1st Edition</name>
        <author>Kyle Simpson</author>
    </book>
</catalog>
```

#### Näide XML'ist

Kõigil suurematel veebilehitsejatel on sisseehitatud XML-parser XML-ile juurdepääsuks ja selle muutmiseks.

### Harjutus: XML faili kuvamine lehe.

1. Valmista endale uus *Vanilla* keskkond [Code Sandbox](#)'is.

2. Kasuta järgmist malli ***index.js*** sees:

### ***index.js***

```
document.getElementById("app").innerHTML = `<table id="xmlTable"></table>`;
```

3. Loo **src** kausta XML fail näiteks on toodud mängude kohta, aga mõtle endale muu teema. Vaheta küsimärgid enda mängu valiku vastu:

### ***games.xml***

```
<?xml version="1.0" encoding="UTF-8"?>
<gameslist>

<game>
  <title lang="en">Hearthstone</title>
  <price>Free</price>
</game>

<game>
  <title lang="en">???</title>
  <price>???</price>
</game>

<game>
  <title lang="en">???</title>
  <price>???</price>
</game>

</gameslist>
```

4. Lisa XML faili saamiseks funktsioon. Vaheta küsimärgid ***games.xml*** allikaga:

### ***index.js***

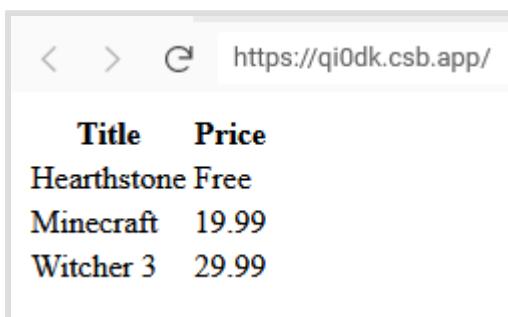
```
xmlhttp.open("GET", "???", false);
xmlhttp.send();
let XMLContent = xmlhttp.responseXML;
```

5. Lisa tabeli generatsiooni funktsioon. Otsi küsimärkide asemele XML failist nimesi:

**index.js**

```
let tableRows = "<tr><th>???</th><th>???</th></tr>";  
let gameElements = XMLContent.getElementsByTagName("???");  
  
for (let i = 0; i < gameElements.length; i++) {  
    tableRows +=  
        "<tr><td>" +  
        gameElements[i].getElementsByTagName("??") [0].childNodes[0].nodeValue +  
        "</td><td>" +  
        gameElements[i].getElementsByTagName("??") [0].childNodes[0].nodeValue +  
        "</td></tr>";  
}  
  
document.getElementById("xmlTable").innerHTML = tableRows;
```

6. Tulemus võiks välja näha selline:



Title	Price
Hearthstone Free	19.99
Minecraft	29.99
Witcher 3	29.99

7. Uuenda XML faili:

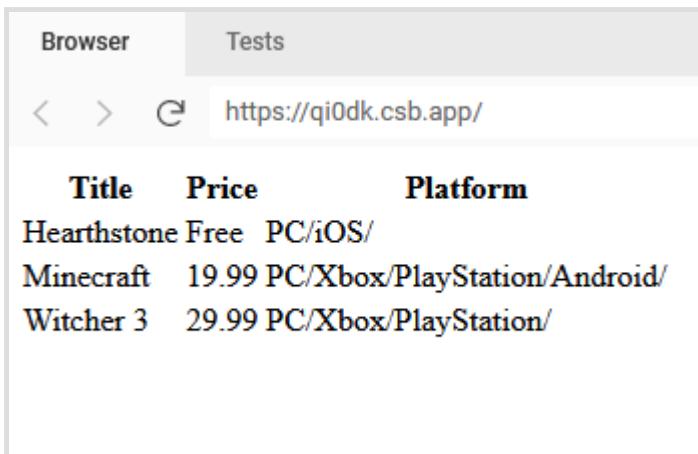
**games.xml**

```

<?xml version="1.0" encoding="UTF-8"?>
<gameslist>
  <game>
    <title lang="en">Hearthstone</title>
    <price>Free</price>
    <platforms>
      <platform>PC</platform>
      <platform>iOS</platform>
    </platforms>
  </game>
  <game>
    <title lang="en">Minecraft</title>
    <price>19.99</price>
    <platforms>
      <platform>PC</platform>
      <platform>Xbox</platform>
      <platform>PlayStation</platform>
      <platform>Android</platform>
    </platforms>
  </game>
  <game>
    <title lang="en">Witcher 3</title>
    <price>29.99</price>
    <platforms>
      <platform>PC</platform>
      <platform>Xbox</platform>
      <platform>PlayStation</platform>
    </platforms>
  </game>
</gameslist>

```

8. Nüüd muuda tabeli loomise koodi, lisades tabelile juurde ühe **header**-i koos **data** sisuga.
9. Lõpuks, lehel võiks olla 3 pealkirja koos sisuga, näiteks:



The screenshot shows a browser window with a header bar containing 'Browser' and 'Tests'. Below the header is a navigation bar with back, forward, and refresh buttons, and a URL field showing 'https://qi0dk.csb.app/'. The main content area displays a table with the following data:

Title	Price	Platform
Hearthstone	Free	PC/iOS/
Minecraft	19.99	PC/Xbox/PlayStation/Android/
Witcher 3	29.99	PC/Xbox/PlayStation/

## 2.3 Rakenduse liidestamine andmeallikatega

Selleks, et rakendus oleks kasulik, peab see andmeid töötlemma. Vaatame lähemalt, millist tüüpi andmeallikaid on olemas.

Erinevad andmeallikad, millega rakendust liidestada;

Objekte loetakse JSON ja XML allikatest, kas neile vastava lõpuga failide seest (js, xml) või läbi REST HTTP päringu. JavaScript'i massiivid käivad ka andmeallikate alla, kuid **array** massiivid on enamjaolt kasutusel JavaScript failides töötlemisel.

JSON andmeid on väga lihtne töödelda. Veebis kasutatakse üldiselt JSON-nit, kuna see on lühem ja väiksem suuruse poolest kui XML ning on väga populaarseks saanud REST teenuste puul.

CSV faili kasutatakse *spreadsheet*'ide (Excel) ja andmebaaside puhul importimisel ja eksportimisel, näiteks et oleks kerge ja kiire andmeid ühest keskkonnast teise viia. CSV on kasutusel tabel tüüpi andmete jagamisel.

### Kontrollküsimused

1. Mis on objekti ja massiivi vahe?
2. Mis on JSON ja XML vahe?
3. Millal kasutada CSV faili tüüpi andmete jagamisel?

## 2.4 Teiste süsteemidega liidestumise strateegiad ja ohud

Kui liidestad enda süsteemi teise süsteemiga, on oht, et omanik muudab oma süsteemi sisu nii, et liidestus enam ei tööta. Tavaliselt teavitatakse sellest pikalt ette ja dokumenteeritakse seda, kuidas süsteeme uuendada, et nad teatud kuupäevast edasi töötaksid. Näide:

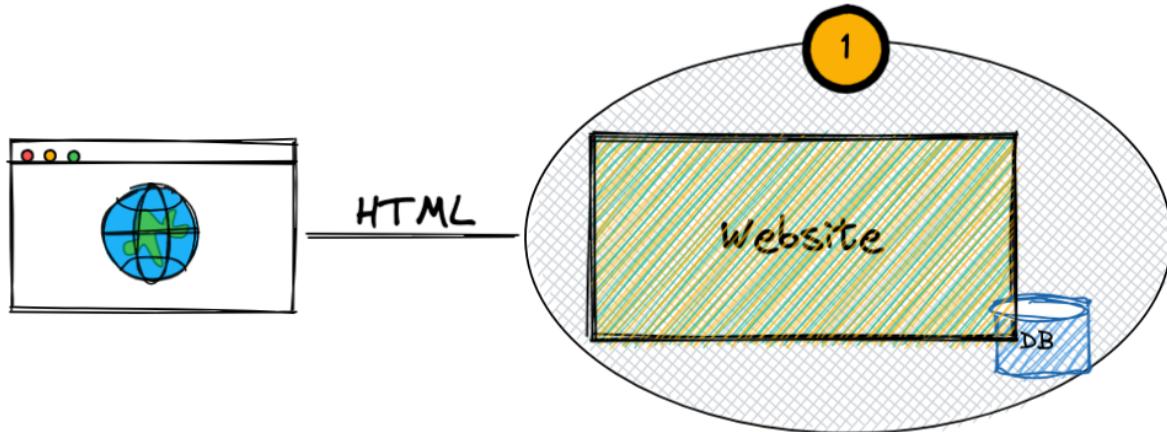
<https://web.archive.org/web/20211212231035/https://developers.facebook.com/blog/post/2021/06/28/deprecating-support-fb-login-authentication-android-embedded-browsers/>

- Sa ei pruugi teada kui turvaline on süsteem, mida kasutad.
- "Shadow API"-d ehk ilma nõuetekohase järelevalve või heaksikiiduta loodud API'd on ohtlikud kui keegi ei tea nende olemasolust. Ülejäänud testimise lõpp-punktid ja domeenid võivad olla avalikult kättesaadavad ilma kellegi teadmata.
- Andmete kokkupuude tekib siis, kui API'd tagastavad kliendile liiga palju andmeid. Iga klient peaks saama ainult need andmed, mida ta vajab oma funktsiooni täitmiseks. Vastasel juhul võib tekkida kaitse nõrkus, mis paljastab API või muud tundlikud andmed.

## 2.5 Võrgurakenduste (client-server, peer-to-peer) aluste ülevaade;

### Monoliitne arhitektuur

**Monoliitseks arhitektuuriks** nimetatakse rakendust, mis ei ole jagatud eraldiseisvateks osadeks, vaid koosneb ühest suurest tervikust. Monoliitne arhitektuur on hajusrakenduse vastand.



*Monoliitne ehk ühest komponendist koosnev rakendus*

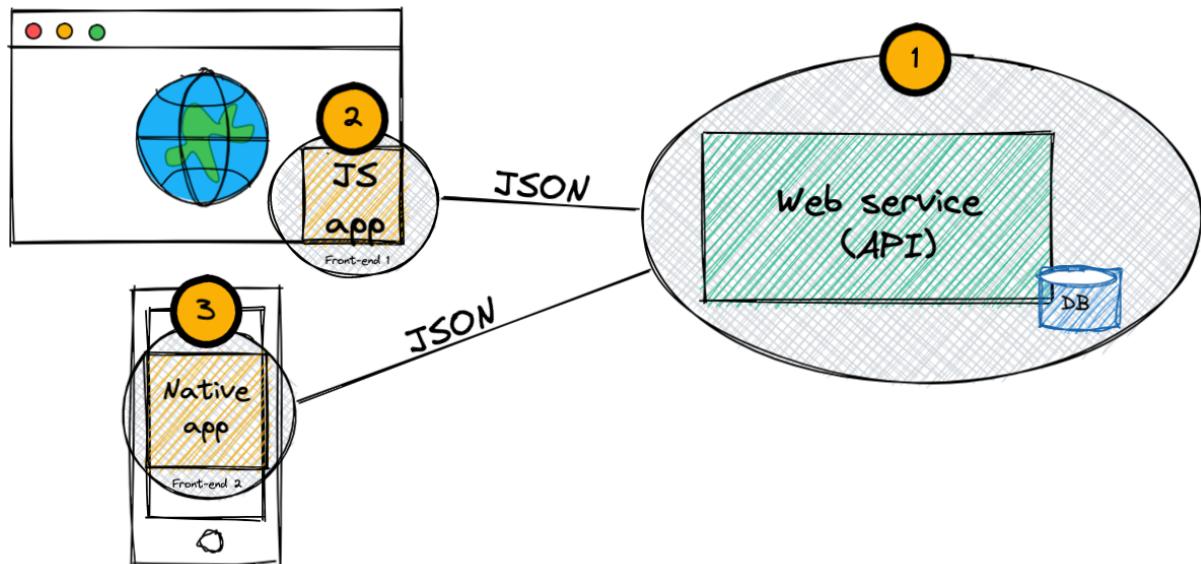
**Kasutada** prototüüpimise ja väiksemat sorti projektide puhul (nt kui [SaaS](#) firma on alles idufirma), sest siis on veel vaja ainult ühte keskset süsteemi, mis võimaldaks ärieesmärke kiiresti ja odavalt saavutada ja keerukamatid süsteemid kulutaks tarbetult aega ja raha.

#### Eelised:

- lihtne ja väga kiire arendustsükliga
- lihtne hallata
- kiire lehe esialgne laadimine

## Klient-server arhitektuur

**Klient-server-arhitektuurimudeli** korral on rakenduse osad omaette rakendused, mis kasutavad suhtlemiseks rakenduse programmeerimise liidest ehk API-it: kliendid (front-endid) saadavad serverile ehk back-endile pärtinguid (taotlusid saada andmeid) ja saavad vastu masintöödeldavaid andmestruktuure (XML või JSON formaadis).



Kolmest komponendist koosnev rakendus ehk hajusrakendus

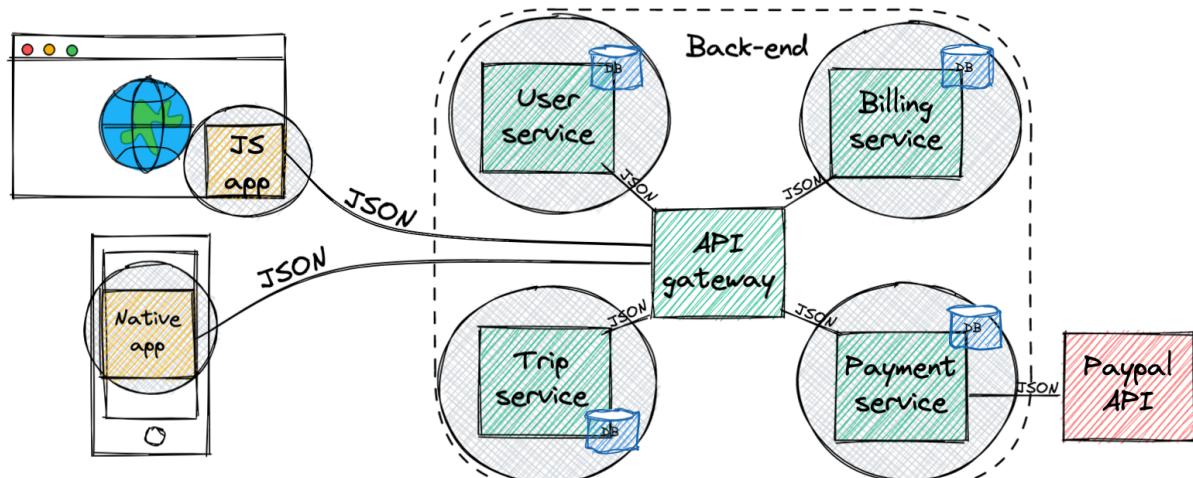
Kasutada näiteks chat või email rakenduste puhul.

Eelised:

- taaskasutus
- API on "tasuta"

## Mikroteenuste arhitektuur

**Mikroteenuste arhitektuuris** on rakendus jagatud väikesteeks tükikesteks - teenusteks -, mida saab eraldi skaleerida ja asendada.



Kasutada kui on arendajaid on palju või ühe serveri ressurssidega ei vea enam välja.

### Eelised:

- selgepiiriline tööjaotus ja vastutusala, kui arendustiime on mitu
- väga hea skaleeritavus
- lihtsam veatuvastus
- programmeerimiskeele ja operatsioonisüsteemi vabadus
- võimalus rakendust uuendada osade kaupa

### Peer-to-peer

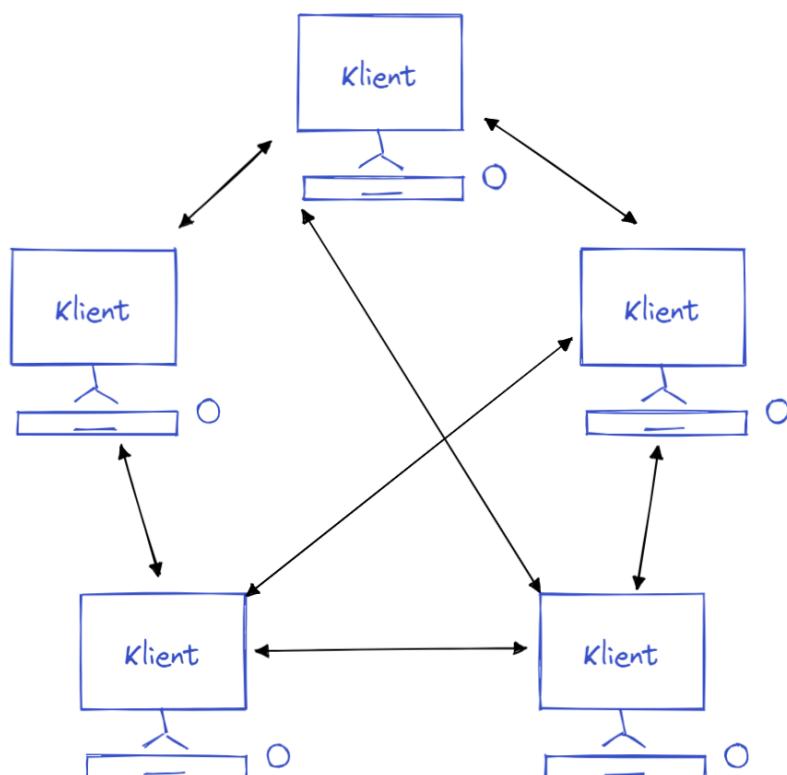
**Peer-to-peer** (P2P) võrk tekib, kui kaks või enam arvutit on ühendatud ja jagavad ressursse ilma eraldi serverit läbimata. See on lihtsat tüüpi võrk, kus arvutid saavad omavahel suhelda ja jagada oma arvutis olevaid või sellega seotud andmeid teiste kasutajatega.

### Eelised:

- Odav
- Pole vaja võrguhaldurit
- Lihtne käsitleda
- Väike võrguliiklus

### Puudused:

- Ei saa tsentraalselt varundada faile ja kaustu
- Aeglustab jõudlust
- Turvalisus



Joonis: peer-to-peer mudel

#### Kontrollküsimused

1. Millise arhitektuurimudeli korral kasutavad rakendused API-t?
2. Millal tuleks üle minna monoliitselt arhitektuurimudelilt mikroteenuste arhitektuurimudeli peale?
3. Millises mudelis on iga arvuti nii server kui ka klient?

### 2.6.1 TCP/UDP transpordiprotokollid ja nende kasutamine;

#### TCP

TCP (Transmission Control Protocol) on ühendusele orienteeritud sideprotokoll, mis hõlbustab sõnumite vahetamist arvutiseadmete vahel võrgus. See on kõige levinum protokoll võrkudes, mis kasutavad interneti-protokolli (IP). Mõnikord nimetatakse neid koos TCP/IP-ks. TCP võtab rakendusest/serverist sõnumeid ja jagab need pakettideks, mida võrgu seadmed saavad seejärel sihtkohta edastada. TCP nummerdab iga paketi ja paneb need enne rakenduse/serveri vastuvõtjale üleandmist uuesti kokku. Kuna see on ühendusele orienteeritud, tagab see ühenduse loomise ja säilitamise, kuni sõnumi andmevahetus on lõpule viidud.

#### UDP

Reaalajas toimivate teenuste, näiteks arvutimängude, kõne- või videovahetuse jaoks on vaja UDP (User Datagram Protocol). Kuna on vaja suurt jõudlust, lubab UDP viivitusega pakettide töötlemise asemel pakette maha jäätta. UDP-s puudub veakontroll, seega sääästab see ka ribalaiust. UDP on töhusam nii latency kui ka bandwidth poolest, kuid kuna veakontroll puudub, ei saa UDP puhul tagada, et kõik andmed 100%-lt on kohale jõudnud. Kui videokönes tekib sekundi murdosa jooksul mõni anomalia ekraanil, pole see probleem, aga kui faile kopeerida ja mõni bait failis on puudu, ei ole see vastuvõetav. Seetõttu ei saa UDP-d igas olukorras kasutada.



*Video voogedastuse artefakt, mis on tingitud veakontrolli puudumisest (Allikas: <https://video.stackexchange.com/questions/24725/grime-like-blocking-compression-artifacts-on-the-decompressed-video-h-264>)*

Seda, millised ühendused arvutil hetkel lahti on ja mis tüüpi nad on, saab vaadata `netstat` käsuga käsurealt.

## 2.6.2 HTTP protokoll, messages, request-response.

### HTTP protokoll

HTTP (Hypertext Transfer Protocol) on protokoll ressursside, näiteks HTML dokumentide kätesaamiseks. See on mis tahes andmevahetuse alus veebis ja see on klient-server protokoll, mis tähendab, et päringud algatab vastuvõtja (veebibrauser). HTTP-päringute ja -vastuste õige vorming sõltub HTTP-protokolli versioonist, mida klient ja server kasutavad. HTTP-protokolli versioonid, mida internetis tavaliselt kasutatakse on HTTP/1.0. See versioon on varasem protokoll ja sisaldab vähem funktsioone. HTTP/1.1 versioon on hilisem protokoll ja sisaldab rohkem funktsioone. Klient ja server võivad kasutada HTTP-protokolli erinevaid versioone. Nii klient kui ka server peavad teatama oma taotluse või vastuse HTTP-versiooni oma sõnumi esimeses reas.

### Messages

HTTP-sõnumid on viis, kuidas andmeid vahetatakse serveri ja kliendi vahel. Sõnumeid on kahte tüüpi: kliendi poolt saadetud taotlused (requests), mis käivitavad mingi tegevuse serveris, ja vastused (responses), mis on serveri vastus. Arendajad koostavad neid HTTP tekstisõnumeid harva: seda teeb tarkvara, veebilehitseja, proxy või veebiserver. Need pakuvad HTTP-sõnumeid konfiguratsiooni failide (proksi või serverite puhul), API-de (brauserite puhul) või muude liidestega kaudu.

## Request

HTTP päringu teeb klient nimelisele hostile, mis asub serveris. Päringu eesmärk on pääseda ligi serveris olevale ressursile. HTTP päringu anatoomia leiab [siit](#).

HTTP pärning sisaldab järgmisi elemente:

- päringurida
- rida HTTP-pealkirju ehk päisevälju
- vajaduse korral sõnumi põhiosa

HTTP määratleb hulga päringumeetodeid, et näidata soovitud toimingut, mida konkreetse ressursi puhul soovitakse teha. Põhilised päringumeetodid leiab peatükis [“Kuidas erineb käsu \(ehk tegevuse, mida soovitakse teha\) edastamine?”](#)

## Response

HTTP-vastuse (response) annab server kliendile. Vastuse eesmärk on anda kliendile tema poolt taotletud ressurss või teavitada klienti sellest, et tema poolt taotletud tegevus on teostatud või taatluse töötlemisel on tekkinud viga.

HTTP-vastus sisaldab:

- staatuse rida (protokolli versioon nt. `HTTP/1.1`, staatuskood nt. `404` ja staatuse tekst `Not Found`)
- rea HTTP-pealkirju ehk päise välju (nt. `Connection: Keep-Alive`)
- sõnumi keha, mis on tavaliselt kohustuslik (nt. `Hello world!`)

### Kontrollküsimused

1. Kas klient ja server saavad kasutada erinevaid HTTP-versioone päringute tegemisel?
2. Millised on HTTP-sõnumi tüübid?
3. Mis toimingut saab teha PATCH päringumeetodiga?

## 2.7 Lihtsad RPC API-d

RPC-põhised API'd on suurepärased toimingute (st protseduuride või käskude) jaoks.

REST kasutab selliseid HTTP-meetodeid nagu GET, POST, PUT, DELETE, OPTIONS ja ka PATCH, et anda toimingu kavatsuse semantiline tähendus. RPC seda aga ei tee. Enamik kasutab ainult GET ja POST, kusjuures GET'i kasutatakse teabe hankimiseks ja POST'i kõige muu jaoks. Tavaliselt näeb RPC API'sid, mis kasutavad midagi sellist nagu POST /deleteFoo, mille keha on `{ "id": 1 }`, selle asemel, et kasutada REST lähenemist, mis oleks DELETE /foos/1.

CRUD süsteemil RPC-sõnumi vastus saatmine POST-meetodi abil:

```
{"userId": 501, "message": "Hello!"}
```

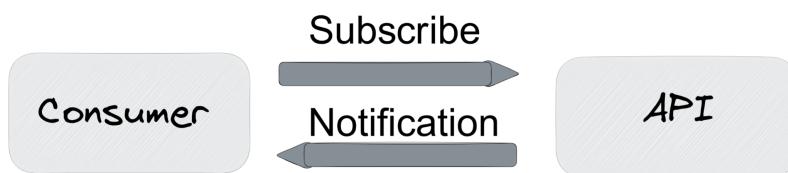
Aga REST'i puhul:

```
{"message": "Hello!"}
```

## 2.8 Event-driven API's: WebHooks, WebSockets, HTTP streaming

Sündmuspõhine API peab pakkuma oma tarbijatele järgmist:

- mehhanism, mis võimaldab tarbijatel tellida neile huvipakkuvaid sündmusi
- tellijatele sündmuste edastamine asünkroonselt



Joonis: Tellimise sündmuspõhine funktsioon

Nende põhjal võime määratleda sündmusepõhiste APIde jaoks järgmise interaktsionimudeli.

### WebHook

WebHook on avalikult juurdepääsetav HTTP POST lõpp-punkt, mida haldab sündmuse tarbija (event listener). Teenusepakkija kes toetab WebHooki võimaldab saata HTTP päringu kindla sündmuse käivitumise (event dispatch) korral sinu valitud sündmuse tarbija aadressile. Näiteks e-poe puhul on kasulik kui saada pangalt maksekinnitus, seda võib teha läbi WebHooki. E-pood registreerib panga administraatori liideses enda sündmuse tarbija aadressi "<https://pood.ee/api/bank-hook>" ja seob selle sündmusega "Laekumine kontole EE7700117772211." Niipea kui kontole laekub raha, saadab pank "<https://pood.ee/api/bank-hook>" aadressile digiallkirjastatud tehingu kokkuvõtte koos summa ja selgitusega. Nii saab e-pood aru milline klient mis tellimuse eest on tasunud.

### WebSocket

Erievalt WebHooks'ist võimaldab WebSocket protokoll pidevat kahesuunalist suhtlust serveri ja kliendi vahel, mis tähendab, et mölemad pooled saavad suhelda ja vahetada andmeid vastavalt vajadusele. WebSocket sobib reaalajas andmehetuseks.

#### Kontrollküsimused

1. Mis on WebHookil ja WebSocketil ühist?
2. Mis on WebHookil ja WebSocketil erinevat?

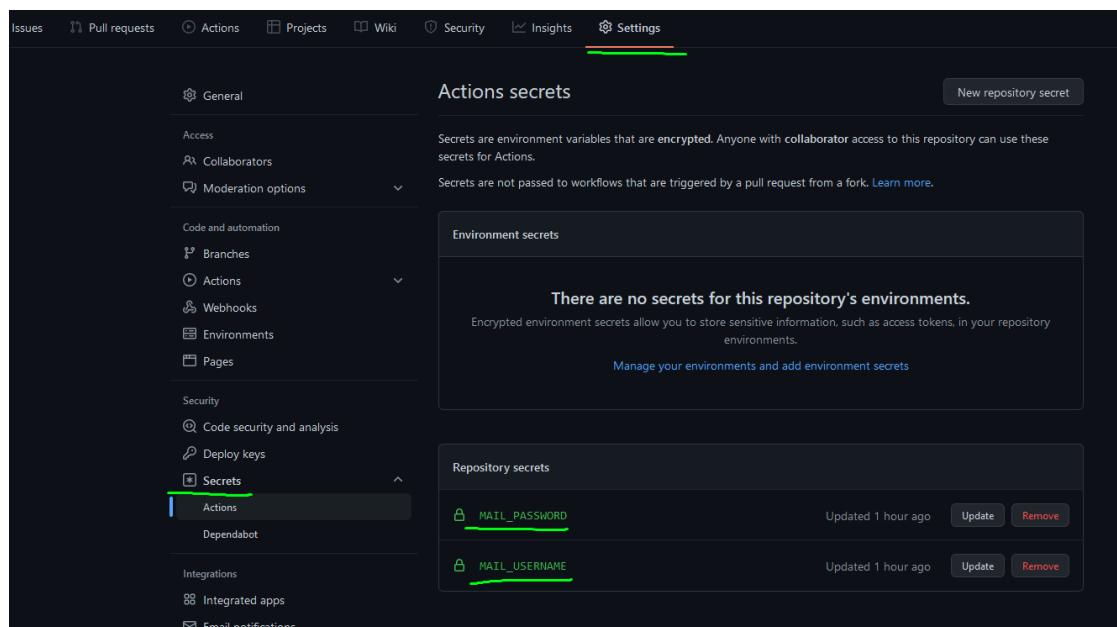
## 2.9 Andmete eksportimine WebHookidega

Harjutus: Saada email Github *push*-imisel.

1. Looge tühi repo.
2. Kloonige see repo endale arvuti.
3. Loo repo kausta kaks kausta struktuuriga: ".github/workflows"
4. Loo *workflows* kausta YAML fail. Pane nimeks näiteks **mail-on-push.yml**.
5. Lugege Github *Actions* struktuurist YAML failide puhul siit:  
<https://docs.github.com/en/actions/quickstart>
6. Lisage dawidd6/action-send-mail Github e-maili saatja YAML faili:

```
jobs:  
  mail_on_push:  
    runs-on: ubuntu-latest  
    steps:  
      - name: Send mail  
        uses: dawidd6/action-send-mail@v3
```

7. Lisage Github **Secrets** alla **MAIL\_PASSWORD** ja **MAIL\_USERNAME**



8. Lisage meili saatjale vajalikud nõuded:

```
with:  
  server_address: smtp.gmail.com  
  server_port: 465  
  username: ${{secrets.MAIL_USERNAME}}  
  password: ${{secrets.MAIL_PASSWORD}}
```

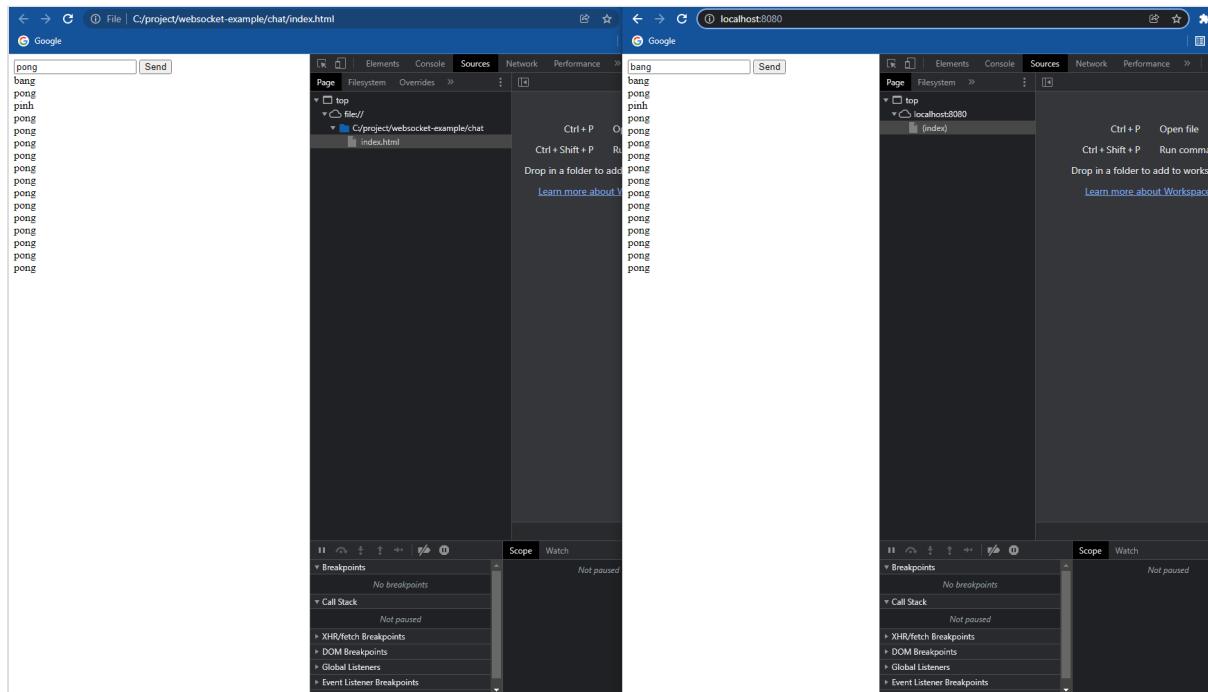
- Lisage nõuetele meili pealkiri koos sisuga, sõnum koos sisuga ja kellele see meil saadetakse

```
subject: ~
body: ~
to: ~
from: Github actions
```

- Et saada *push*-itud repositooriumi nime, kasutage `${{github.repository}}`. Kehtestuse loaja nime jaoks `${{github.event.pusher.name}}` ja kehtestuse sõnumi jaoks `${{github.event.head_commit.message}}`.
- Lisage `on: [push]` faili, et meil saadetakse kehtestuse *push*-imisel.
- Proovige järelle: tehke *push* kaustas, kus asub `.github/workflows` ja selles kaustas asub eelnevalt tehtud YAML fail. Tulemust näeb *push*-itud repositooriumi *Actions* all..

## 2.10 Reaalaegne andmevahetus WebSocketiga

Harjutus - Vestlusruum serveri ja klientrakenduse vahel.



Joonis: WebSocket ja Node'i vestlus rakendus

- HTML saab kätte siit: <https://pastebin.com/raw/DEm1XAfz>
- Node'i WebSocket'i sisaldamine.

```
1 const http = require('http');
2 const fs = require('fs');
3 const ws = new require( id: 'ws');
```

### 3. Serveri seadistamine

```
5 const wss = new ws.Server({noServer: true});
6
7 function accept(req, res) {
8     if (req.url === '/ws' && req.headers.upgrade &&
9         req.headers.upgrade.toLowerCase() === 'websocket' &&
10        req.headers.connection.match( matcher: /\bupgrade\b/i)) {
11            wss.handleUpgrade(req, req.socket, Buffer.alloc( size: 0), onSocketConnect);
12        } else if (req.url === '/') { // index.html
13            fs.createReadStream( path: './index.html').pipe(res);
14        } else { // page not found
15            res.writeHead(404);
16            res.end();
17        }
18    }
19}
```

### 4. Ühenduse loomine

```
20 const clients = new Set();
21
22 function onSocketConnect(ws) {
23     clients.add(ws);
24     ws.on('message', function(message) {
25         message = message.slice(0, 50); // max message length will be 50
26         for(let client of clients) {client.send(message);}
27     });
28     ws.on('close', function() {
29         log(`connection closed`);
30         clients.delete(ws);
31     });
32 }
33
```

### 5. Teksti kuvamine

```

34     let log;
35
36     if (!module["parent"]) {
37         log = console.log;
38         http.createServer(accept).listen( port: 8080);
39     } else {
40         // to embed into javascript.info
41         log = function() {};
42         // log = console.log;
43         exports.accept = accept;
44     }

```

## 2.11 HTTP streaming – voogedastuse tellimine läbi (CDN)

Tuntud ka kui *Comet* mudelina. *Comet* on veebirakenduse mudel, mille puhul pikalt hoitud HTTPS-päring võimaldab veebiserveril suruda andmeid brauserile, ilma et brauser seda selgesõnaliselt küsiks. *Cometi* voogedastust kasutav rakendus avab kõigi *Cometi* sündmuste jaoks ühe püsiva ühenduse kliendibrauserist serveriga. Neid sündmusi käsitletakse ja tölgendatakse kliendi poolel järk-järgult iga kord, kui server saadab uue sündmuse, kusjuures kumbki pool ei sulge ühendust.

HTTP streaming'u üks eelis on see, et kõik interneti-ühendusega seadmed toetavad HTTP-d, mis muudab rakendamise lihtsamaks kui voogedastuse protokollid, mis nõuavad spetsiaalsete serverite kasutamist. Teine eelis on see, et HLS voog võib suurendada või vähendada videokvaliteeti sõltuvalt võrgutingimustest, ilma et taasesitus katkeks. Seetõttu võib videokvaliteet muutuda paremaks või halvemaks video keskel, kui kasutaja seda vaatab. Seda omadust nimetatakse "adaptive bitrate video delivery" või "adaptive bitrate streaming" ja ilma sellesta võivad aeglased võrgutingimused video esituse täielikult peatada.

HLS on väga hästi skaleeritav videofailide ja voogedastuse sisu edastamiseks ülemaailmse sisu edastusvõrkude (CDN) kaudu. HLS-i saab hõlpsasti skaleerida tavalistele veebiserverite abil üle globaalse sisu edastamise võrgu (CDN). Jagades töökoormust serverite võrgustiku vahel, suudavad CDNid tulla toime oodatust suurema vaatajakonnaga. CDNid aitavad ka parandada vaatajakogemust heli- ja videolõikude vahemälu abil.

### Silmaringi laiendamiseks:

1. Loe adaptiivse bitikiirusega voogesitusest:  
[https://en.wikipedia.org/wiki/Adaptive\\_bitrate\\_streaming](https://en.wikipedia.org/wiki/Adaptive_bitrate_streaming)
  
2. Loe HLS-ist:  
[https://en.wikipedia.org/wiki/HTTP\\_Live\\_Streaming](https://en.wikipedia.org/wiki/HTTP_Live_Streaming)

## 2.12 REST vs GraphQL

2012 aastal töötas Facebooki iOS tiim uue mobiilirakenduse kallal ja neil tekkis raskusi andmete kättesaamisega REST teenusest. Ilmselt ei olnud Facebookis REST teenus korrektsest dokumenteeritud, sest iOS tiimil tekkis palju vigu, mis olid tekkinud sellest, et nad eeldasid valesti, mis kujul andmed serverist tagasi tulevad. Infrastruktuuri tiim, kelle ülesandeks on teiste tiimide tööd lihtsamaks teha, hakkas nende probleeme uurima ja see lõppes GraphQL-i välja arendamisega, mis on alternatiiv REST teenusele.

Erinevus REST teenuse ja Graph QL-i puhul on see, et kui REST-i puhul on etteprogrammeeritud kindlad lõpp-punktid nagu /users või /orders, millele saab saata kas GET, POST, PUT, PATCH või DELETE päringuid ja sealult tulevad kindla fikseeritud andmestruktuuriga vastused, siis GraphQL-i puhul eksisteerib ainult üks lõpp-punkt, millele saab saata POST päringu, küsides ja saades ainult need andmed, mida tegelikult on vaja.

Sest nii mõnigi kord on REST API vastuses palju atribuute, mis sageli ei ole hetkel vajalikud, nii et seadmed, mis on aeglase internetiga, peavad kaua ootama suurt JSON-it, millega nad ainult väikese osa ära kasutavad.

Võrreldes GraphQLiga on vigade käsitelemine (*error handling*) REST-is lihtsam. RESTful API-d järgivad HTTP spetsifikatsiooni ja tagastavad erinevaid HTTP-staatusi erinevate API-taotluse tulemuste jaoks. GraphQL seevastu tagastab iga API päringu, sealhulgas vigade puhul, staatuse `200 OK`. See raskendab integreerimist seirevahenditega (*monitoring tools*).

## Nime ja aadressi küsimine: REST vs GraphQL



(Pildi allikas: <https://hasura.io/learn/graphql/intro-graphql/graphql-vs-rest/>)

Siit on näha, et REST-is tehakse 2 pärингut (üks ressurssi user ja teine ressurssi address), aga GraphQL-is saab samad andmed ühe pärингuga. Iga pärинг võtab sõltuvalt võrgulatentsusest aega (mida saab mõõta käsuga *ping*).

Pange tähele, et antud näites on nimetatud ressursid ainsuses, kuid eelstatud viis on kasutada mitmust ressursi nimedes (*user* asemel *users* ja *address* asemel *addresses*).

### Kontrollküsimused

1. Mis eelised on GraphQL-iil REST-i ees?
2. Mis eelised on REST-iil GraphQL ees?
3. Kas GraphQLi puhul on vaja jälgida pärингu sisu järjekorda?

## 2.13 GraphQL päringukeel + object-store NoSQL andmebaasid (nt. MongoDB, Redis, memcached)

### GraphQL

GraphQL on rakenduste programmeerimisiidest (API) päringukeel, mis seab prioriteediks anda klientidele täpselt need andmed, mida nad taotlevad.

tagastatakse. RESTi kasutades saadakse alati täielik andmekogum, mis ei ole optimaalne.

REST-i puhul peab klientrakendus soovitud tulemuse saamiseks tegema mitu pärингut serverisse. Näiteks sotsiaalvõrgustiku profiililehe andmete saamiseks (isiku nimi, esimesed 6 fotot, esimene 9 sõbra fotod, sõprade arv jne) peaks tegema pärингuid lõpp-punktidele /persons/xxx, /posts?author\_id=xxx, /photos?limit=9&owner\_id=xxx, /persons?friendOf=9.

Võib küll kõrvale pöigata RESTi põhimöttest, et andmed peaks olema organiseeritud ressurssideks ning teha ühe spetsiaalse lõpp-punkti /profile/xxx, mis sisaldab spetsiaalselt täpselt kõike vajalikku, kuid selle peab back-endis spetsiaalselt selle vaate jaoks programmeerima, samas kui osaliselt sama informatsiooni võib vaja minna ka paljudes muudes olukordades.

GraphQL lahendab selle probleemi pakkudes võimalust pärida kõigest ühe päringuga kõik vajalikud andmed:

```
{  
  user(id: 480218) {  
    id  
    name  
    isViewerFriend  
    profilePicture(size: 50) {  
      uri  
      width  
      height  
    }  
    photos(first: 6) {  
      uri  
      width  
      height  
    }  
    friendConnection(first: 9) {  
      totalCount  
      friends {  
        id  
        name  
      }  
    }  
  }  
}
```

GraphQL-vastus on kujundatud täpselt nii, nagu seda kirjeldatakse: see, mida küsiti, ilma muutusteta. RESTi puhul aga ei saa piirata välju, mida konkreetsest lõpp-punktist

## NoSQL andmebaasid

NoSQL-andmebaasid nagu MongoDB, Redis ja Memcached suudavad käsitleda kiiresti muutuvaid struktureerimata andmeid, erinevalt ridade ja tabelitega relatsioonilised (SQL) andmebaasid. Samuti on võimalik olemasolevate NoSQL-töökoormuste puhul lihtne andmebaasid üle via pilve.

NoSQL-andmebaasid suudavad salvestada eri tüüpi andmeid ja ei pea olema nii struktureeritud kui SQL-andmebaasid. Seega võimaldavad mitterelatsioonilised andmebaasid suurt kohanemisvõimet ja paindlikkust, mistõttu on see sobivam valik suurte struktureerimata ja omavahel mitteseotud andmekogumite töötlemisel.

Aja jooksul on välja kujunenud neli peamist NoSQL-andmebaasi tüüpi:

**Dokumendiandmebaasid** salvestavad andmeid dokumentides, mis sarnanevad JSON (JavaScript Object Notation) objektidele. Iga dokument sisaldab väljade ja väärustute paare. Väärtused võivad tavaliselt olla erinevat tüüpi, sealhulgas sellised asjad nagu stringid, numbrid, booleans, massiivid või objektid.

**Võti-väärtusandmebaasid** on lihtsamat tüüpi andmebaasid, kus iga element sisaldab võtmeid ja väärtusi.

**Laia veeruga andmekogud** salvestavad andmeid tabelites, ridades ja dünaamilistes veergudes.

**Graafiandmebaasid** salvestavad andmeid sõlmedes ja servades. Sõlmedes hoitakse tavaliselt teavet inimeste, kohtade ja asjade kohta, samas kui servades hoitakse teavet sõlmede vaheliste suhete kohta.

NoSQL on sobilik:

- Kiiresti muutuvate andmete käsitlemiseks.
- Skeemiga mitteeseotud andmed või rakenduse poolt dikteeritud skeem
- Rakendustele, kus jõudlus ja kättesaadavus on olulisemad kui järjepidevus
- Rakendustele, mis on alati kättesaadaval ja selle kasutajaid on üle kogu maailma

#### Kontrollküsimused

1. Millal võtta kasutusele NoSQL?
2. Millised on NoSQLi omadused?
3. Kas NoSQL on tegemist relatsioonilise või mitterelatsioonilise andmebaasiga?

**ÕV3: Realiseerib mõistlikud meetodid kasutajate tuvastamiseks ja veebiteenuste turvalisuse tagamiseks sh terviklikkus ja käideldavus;**

## 3. VEEBITEENUSE TURVALISUSE TAGAMINE KASUTADES SOOVITATUD PRAKTIKAID

### 3.1 OpenID, OAuth, JSON Web Token (JWT)

#### OpenID

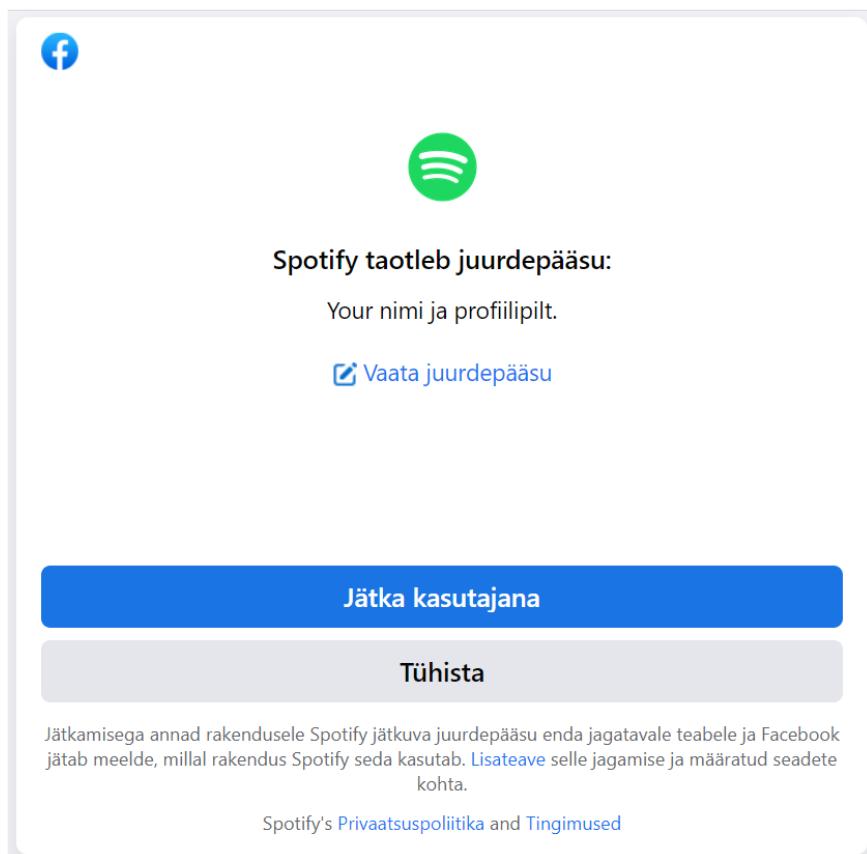
OpenID on protokoll, mis võimaldab veebisaitidel või rakendustel anda kasutajatele juurdepääsu, autentides neid teise teenuse või teenusepakkuja kaudu.

OpenID võimaldab kasutada OpenID teenusepakkuja (näiteks Google) sisselogimise andmeid, et logida sisse teise rakendusse (näiteks Facebooki). Näiteks kui soovite pääseda veebilehele [www.example.com](http://www.example.com), võivad nad küsida teie OpenID-i Google'i või Facebooki kontole kujul. Kui te sisestate oma OpenID, suunab [example.com](http://example.com) teid oma OpenID-teenuse pakkujale ja te autentite end, kinnitab oma identiteeti ning seejärel lubatakse teil juurdepääs oma kontole [example.com](http://example.com)-is.

## OAuth

OAuth (Open Authorization) võimaldab väljastada kolmandatele osapooltele access tokeneid omaniku heaksikiidul. Seejärel saab kolmas osapool kasutada tokenit, et pääseda ligi serveri poolt hallatavatele kaitstud ressurssidele.

Näiteks teil on kunagi palutud mõne rakenduse poolt luba anda juurdepääsuks oma isikuandmetele, näiteks teie Facebooki või Google'i kontaktide jagamiseks, siis olete tõenäoliselt kasutanud OAuthi.



Pilt: Spotify taotleb juurdepääsu Facebooki andmetele

## JSON Web Token (JWT)

JSON Web Token (JWT) on standard, mis määratleb kompaktse ja iseseisva viisi andmete turvaliseks edastamiseks osapoolte vahel JSON-objektina. Teavet saab kontrollida ja usaldada, sest see on digitaalselt allkirjastatud. On olemas ka JSON [debugger](#).

JSON Web Token struktuur:

- Päis (Header)

Päis koosneb tavaliselt kahest osast: märgendi tüüp, mis on JWT ja allkirjastusalgoritm, näiteks HMAC-SHA256 või RSA

- Andmed (Payload)

Andmed koosnevad nõuetest (claims). Nõuded on avaldused üksuse (tavaliselt kasutaja) ja täiendavate andmete kohta. Nõudeid on kolme tüüpi: registreeritud, avalikud ja privaatsed nõuded

Registreeritud nõuded: Need on hulk eelnevalt määratletud nõudeid, mis ei ole kohustuslikud, kuid mida soovitatakse, et pakkuda kasulikke, koostalitusvõimelisi nõudeid. Mõned neist on: iss (issuer), exp (expiration time), sub (subject), aud (audience) jt

Avalikud nõuded: Need võivad JWT-d kasutavate isikute poolt suvaliselt määratletud olla. Kuid kokkupõrgete välimiseks tuleks need määratleda IANA JSON Web Token Registry's või määratleda URI-na, mis sisaldab kokkupörkekindlat nimeruumi.

Privaatsed nõuded: Need on kohandatud nõuded, mis on loodud teabe jagamiseks osapoolte vahel, kes nõustuvad nende kasutamisega, ja mis ei ole registreeritud ega avalikud nõuded

- Allkiri (Signature)

Allkirja kasutatakse selleks, et kontrollida, et sõnumit ei ole vahepeal muudetud.

Allkirja osa loomiseks tuleb võtta kodeeritud päis, kodeeritud andmed, secret, päises määratud algoritm ja allkirjastada see

Kontrollküsimused

1. Mis väljastatakse OAuth-i puhul, et kolmandad osapooled saaksid ressurssidele ligipääsu?
2. Kui lood enda veebilehele sisselogimise vormi ning tahad lisada võimaluse, et kliendid saaksid sisse logida ka Apple ID-ga, siis millist praktikat peaksid kasutama?
3. Millistest osadest koosneb JWT?

### 3.2 HTTP authentication header and cookies. Cookie attributes: Secure, HttpOnly, SameSite

#### HTTP authentication header

HTTP päringu kontekstis on põhiline juurdepääsu autentimine meetod, mille abil HTTP kasutaja (nt veebibrauser) esitab päringu esitamisel kasutajanime ja parooli. Põhilise HTTP-autentimise puhul sisaldab taotlus päise väljal `Authorization: basic`

`<credentials>`, kus credentials on volituste ja parooli kodeering, mis on ühendatud kooloniga :

Authorization header saadetakse tavaliselt, pärast seda, kui kasutaja üritab esimest korda taotleda kaitstud ressurssi ilma autoriseerimata. Server vastab sõnumiga 401 Unauthorized, mis sisaldab vähemalt ühte WWW-Authenticate-päist. See päis näitab, milliseid autentimisskeeme saab ressursile juurdepääsuks kasutada (ja mis tahes lisateavet, mida klient vajab nende kasutamiseks). Kasutaja peaks valima esitatud autentimisskeemidest kõige turvalisema, mida ta toetab, küsimäga kasutajalt tema volitusi ja seejärel taotlema ressurssi uesti (koos kodeeritud volitustega Authorization-pealkirjas).

HTTP autoriseerimistaotluse päiseid saab kasutada selleks, et anda volitused, mis autendivad kasutajaagenti serverile, võimaldades juurdepääsu kaitstud ressursile.

## Cookies and attributes

HTTP-küpsis on väike andmestik, mille server saadab kasutaja veebibrauserile. Brauser võib küpsise salvestada ja saata selle hilisemate pärингute korral samale serverile tagasi. Tavaliselt kasutatakse HTTP-küpsist selleks, et tuvastada, kas kaks pärингut tulevad samast brauserist, näiteks kasutaja sisselogimise säilitamiseks. See jätab meelde olekuteabe olekuta HTTP-protokolli jaoks.

Küpsiseid kasutatakse peamiselt kolmel eesmärgil:

- Seansside haldamiseks (sisselogimised, ostukorvid, mängutulemused jne)
- Isikupärastamiseks (kasutaja eelistused, teemad ja muud seaded)
- Kasutaja käitumise salvestamiseks ja analüüsimeks

## Secure

Secure atribuut on valik, mille rakendusserver saab määrata kui ta saadab kasutajale uue küpsise HTTP-vastuse raames. Secure atribuudi eesmärk on vältida küpsiste jälgimist kõrvaliste isikute poolt, kuna küpsised edastatakse selge tekstina. Brauser ei saada secure atribuudiga küpsist üle krüpteerimata HTTP päringu.

## HttpOnly

HttpOnly on brauseri küpsisele lisatud tag, mis takistab kliendipoolsete skriptide juurdepääsu andmetele. Küpsise ligi saab ainult läbi serveri. See aitab vältida pahatahtliku koodi abil andmete saatmist nt XSS-rünnakuid.

## SameSite

SameSite takistab brauseril selle küpsise saatmist koos cross-site pärингutega. Peamine eesmärk on vähendada cross-origin vahelise teabe lekkimise ohtu. Samuti pakub see kaitset cross-site päringu võltsimise rünnakute vastu. Võimalikud väärтused on tühi (none), lahtine (lax) või range (strict).

Range väärтus takistab küpsise saatmist veebilehitseja poolt sihtkohale kõigis saidiülesel sirvimise kontekstides, isegi kui järgitakse tavalist linki. Näiteks GitHubi sarnase veebisaidi

puhul tähendaks see, et kui sisselogitud kasutaja järgib ettevõtte arutelufoorumis või e-kirjas avaldatud linki GitHubi privaatprojektile, ei saa GitHub seansiküpsist ja kasutaja ei pääse projektile ligi.

Lax väärthus tagab mõistliku tasakaalu turvalisuse ja kasutatavuse vahel veebisaitide jaoks, mis soovivad säilitada kasutaja sisselogitud seanssi pärast seda, kui kasutaja saabub väliselt lingilt. Ülaltoodud GitHubi stsenaariumis oleks sessiooniküpsis lubatud, kui väliselt veebisaidilt järgitakse tavalist linki, kuid CSRF-ohtlike päringumeetodite (nt POST) puhul oleks see blokeeritud.

Tühi väärthus ei anna mingit kaitset. Brauser kinnitab küpsised kõigis saidiülese sirvimise kontekstides.

#### Kontrollküsimused

1. Mis eesmärgil kasutatakse küpsiseid?
2. Mida peatab Secure attribuut HTTP päringul?
3. Mis on SameSite 3 võimalikku väärust?

## 3.3 Session/token expire.

### Session

Kuna HTTP on olekuta, siis selleks, et seostada üks taotlus mis tahes teise taotlusega, on vaja viisi, kuidas salvestada kasutaja andmeid HTTP-päringute vahel.

Küpsised või URL-parameetrid on mõlemad sobivad viisid andmete edastamiseks 2 või enama taotluse vahel. Need ei ole aga head juhul, kui te ei soovi, et need andmed oleksid kliendi poolel loetavad/muudetavad.

Lahendus on näiteks salvestada need andmed serveri poolel, anda neile id ja lasta kliendil teada ainult seda id-d.

### Session expire

Session expire on sündmus, kui kasutaja ei tee veebilehel mingi (veebiserveri poolt määratud) ajavahemiku jooksul ühtegi toimingut. Muudetakse serveri poolel kasutaja sessiooni staatus "kehtetuks" ja antakse veebiserverile korraldus see hävitada (kustutades kõik sessioonis sisalduvad andmed).

### Token expire

Access token väljastatakse kolmandatele isikutele autoriseerimisserveri poolt omaniku loal. Klient kasutab access tokenit, et pääseda ligi serveri poolt hallatavatele kaitstud andmetele.

Refresh token on volitused, mida kasutatakse access tokenite saamiseks. Refresh tokenid väljastatakse kliendile autoriseerimisserveri poolt ning mida kasutatakse uue access tokeni saamiseks kui praegune access token muutub kehtetuks või aegub, või täiendavate access tokenite saamiseks.

Maksimaalse turvalisuse ja paindlikkuse tagamiseks on märkide andmisel levinud meetodiks access tokens ja refresh tokens kombinatsioon, mis tagab maksimaalse turvalisuse ja paindlikkuse.

Kontrollküsimused

1. Miks luuakse sessioone?
2. Millist turvalisust pakub sessiooni aegumine?
3. Mis on *refresh token*?

### 3.4 Kuidas genereerida turvaline räsi.

Räsi genereerimine on krüptograafiline protsess, mida saab kasutada eri tüüpi sisendite autentsuse ja terviklikkuse valideerimiseks. Räsi kasutatakse laialdaselt autentimissüsteemides, et vältida lihtkirjas paroolide salvestamist andmebaasides, kuid seda kasutatakse ka failide, dokumentide ja muud liiki andmete valideerimiseks. Kui räsi on liiga lühike, siis on võimalik kõik variandid ammendavalt läbi proovida ning teha neist tabeli, nn vikerkaaretabeli ([Rainbow Table](#)).

Selleks, et räsi oleks veelgi turvalisem, lisatakse räsile [sool](#).

Ideaalis peaks iga parooli räsimise eksemplar kasutama oma räsifunktsiooni. Samuti ei tohiks sool olla liiga lühike, sest siis on ründajal võimalik genereerida sõnastik iga võimaliku soola jaoks.

#### Harjutus

1. Ava enda koodiredaktor
2. Tee uus fail nimega `generateHash.js`

3. Lisa sinna järgnev kood:

```
1 const bcrypt = require('bcrypt');
2 const myPassword = '';
3
4 console.time( label: 'Time to generate salt');
5 const salt = bcrypt.genSaltSync( rounds: 10);
6 console.log('This is your salt: ' + salt);
7 console.timeEnd( label: 'Time to generate salt');
8
9 console.time( label: 'Time to generate hash');
10 const hashedPassword = bcrypt.hashSync(myPassword, salt);
11 console.log(myPassword + ' is your password & this is your
12 password after hashing it: ' + hashedPassword);
13 console.timeEnd( label: 'Time to generate hash');
```

4. Paigaldada bcrypt käsuga `npm install bcrypt`
5. Anna real 2 muutuja `myPassword` vääruseks mingi tekst, mis on sinu parooliks
6. Käivita fail parem hiireklöps faili sees ning *Run 'generateHash.js'* või kiirklahviga Ctrl + Shift + F10
7. Muuda `genSaltSync` parameetris rounde ning vaata, mis juhtub

Seletus:

- Real 1 laaditakse sisse bcrypt-i, mida kasutatakse paroolide krüpteerimiseks
- Real 5 loob funktsioon `genSaltSync` soola, mida kasutame real 10 räsi loomiseks. Funktsiooni sees parameeter `rounds` (voorud) tähendab tegelikult kulutegurit. Kulutegur kontrollib, kui palju aega kulub ühe räsi arvutamiseks. Mida suurem on kulufaktor, seda rohkem on räsi voorusid. Vaikimisi on voorude arv 10, mis võtab kuskil 0,5 - 1 sekundit aega. Voorude arv 20 võib aga võtta juba ligi terve minuti aega.
- Real 10 loob funktsioon `hashSync` räsi, koos soolaga (mille genereerisime real 5).
- `console.time` ja `console.timeEnd` abil mõõdame funktsioonide aega. `console.time` ja `console.timeEnd` on paaris ning neil peab olema sama sisu (`label`), et leida üles paariline, kus algab või lõpeb aja mõõtmine.

### 3.5 HTTPS ja kuidas genereerida sertifikaat kasutades Let's Encrypt'i

#### HTTPS

Arvatavasti oled kohanud lühendit SSL. See tähendab Secure Sockets Layer ja lühidalt öeldes on see tehnoloogia, mis hoiab internetiühenduse turvalisena ja kaitseb (krüpteerib) mistahes tundlikke andmeid, mida saadetakse kahe süsteemi vahel, takistades kurjategijatel

lugeda ja muuta edastatud teavet, sealhulgas võimalikke isikuandmeid. Need kaks süsteemi võivad olla server ja klient (näiteks veebipoe sait ja brauser) või server serverile (näiteks rakendus, mis sisaldab isikuandmeid või palgaandmeid).

See tagab, et kasutajate ja saitide vahel või kahe süsteemi vahel edastatavaid andmeid ei ole võimalik lugeda, kasutades selleks algoritme, mis andmeid transiidi ajal krüpteerivad. See takistab häkkeritel ühendust pealt kuulata, sest andmed võivad sisaldada krediitkaardinumbreid, nimesid, aadressi jms.

TLS (Transport Layer Security) on lihtsalt SSL-i uuendatud, turvalisem versioon. Me nimetame oma turvasertifikaate endiselt SSL-iks, sest see on üldisemalt kasutatav termin, kuid kui ostate SSL-sertifikaadi, siis ostate tegelikult kõige ajakohasemad TLS-sertifikaadid.

Eesliide https:// (Hyper Text Transfer Protocol Secure) ilmub URL-is, kui veebisait on turvatud SSL-sertifikaadiga. Sertifikaadi üksikasjad, sealhulgas sertifikaadi väljastanud asutus ja veebisaidi omaniku ärinimi, on nähtavad, kui klõpsate brauseriribal olevale lukusümbolile.

Tegelikult on SSL sertifikaat X.509 formaadis andmestik, mis sisaldab tavalist avalik-privaat võtmepaari avalikku võtit. Privaatvõti on veebiserveri mälus, kui veebiserver töötab ja veebiserver kasutab andmete krüpteerimiseks:

**HTTPS** ehk Hypertext Transfer Protocol **Secure** on protokoll, mille puhul HTTP-andmeid edastatakse krüpteeritult üle transpordikihi turbeprotokolli (TLS). Varem kasutati TLS asemel turvasoklite kihti (SSL), kuid see on tänaseks liiga ebaturvaline.

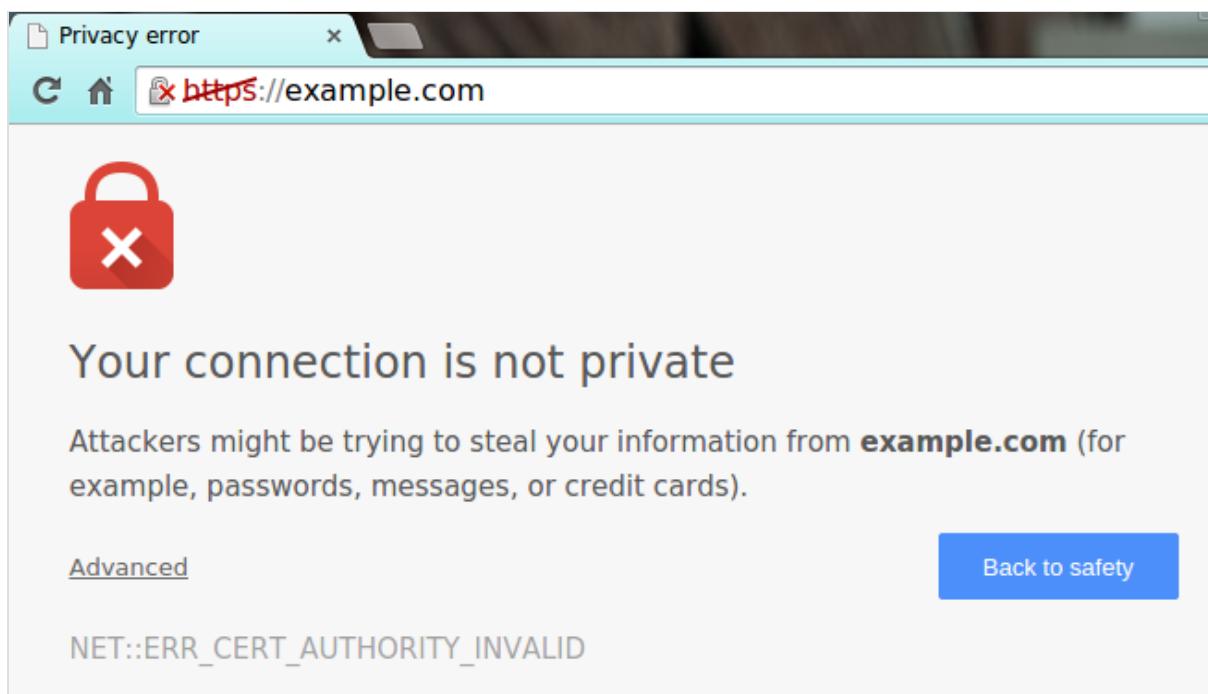
Kui brauser kuvab väikest tabalukku, kasutatakse SSL sertifikaati. SSL (Secure Sockets Layer) on kinnitus veebilehe usaldusvääruse kohta.



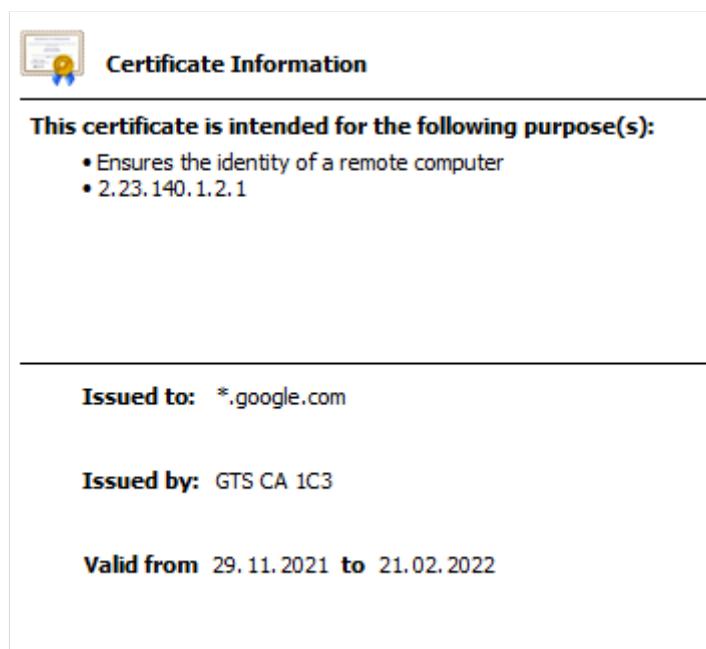
Pilt: Tabalukk Chrome'is

Tabaluku märgile klõpsates saab järele vaadata, kes on sertifikaadi välja andnud ning kui kaua see kehtib.

Sertifikaadi saab ka ise genereerida ja ka siis on andmevahetus krüpteeritud, kuid kuna brauserid ei usalda isetehtud sertifikaate, kuvavad nad esmakordset sellisele veebisaidile minnes hoiatust:



Seetõttu tuleks sertifikaat hankida mõnelt selliselt osapoolelt, mille juurser on operatsioonisüsteemi juursertide hoidlas.



Pilt: Sertifikaat

## Kuidas genereerida sertifikaat?

HTTPS-i saamiseks oma veebisaidile on vaja hankida sertifikaat sertifitseerimisasutusest (CA). Let's Encrypt on sertifitseerimisasutus. Selleks, et saada oma veebisaidi domeenile sertifikaat Let's Encrypt'ilt, tuleb näidata, et on olemas kontroll domeeni üle. Let's Encrypt'i puhul saab seda teha tarkvara abil, mis kasutab ACME-protokolli, mis tavaliselt töötab veebihosteri juures.

Selleks, et välja selgitada, milline meetod kõige paremini sobib, tuleb teada, kas on olemas veebimajutusel ligipääs shell'ile. Kui haldate oma veebisaiti täielikult läbi juhtpaneeli, näiteks cPaneli, Pleski või WordPressi kaudu, on suur töenäosus, et ei ole shell-juurdepääsu.

Link sertifikaadi genereerimisele: <https://www.youtube.com/watch?v=lWBnfpn8aE>

## 3.6 CORS, CSRF, XSS, SQL Injection, Forced browsing

### CORS

CORS (Cross-origin resource sharing) on veebilehitseja mehhanism, mis võimaldab kontrollitud juurdepääsu väljaspool antud domeeni asuvatele ressurssidele. See laiendab ja lisab paindlikkust [SOP](#)-ile (same-origin policy). Samas kaasneb sellega võimalus domeenidevahelisteks rünnakuteks, kui veebisaidi CORS-poliitika on halvasti konfigureeritud ja rakendatud. CORS ei ole kaitse *cross-origin* rünnakute, näiteks *cross-site* päringu võltsimise (CSRF) eest.

Näide mis olukorra eest CORS kaitseb

Meil on kolm osapoole:

- Pank, millel on API
- Bob, panga klient kes kasutab veebilehitsejat, et pangatoiminguid teha
- Charlie, hääkker kes tahab Bobi raha ära varastada ning kelle kodulehel on peidetud ründav JavaScript

Kui Bob külastab Charlie kodulehte (myblog.com) käivitub seal JavaScripti pärning mis proovib alustada pangas teingut. Kui Bobil on teises veebilehitseja aknas või tabis pangas sisse logitud siis veebilehitseja ei näe selles probleemi ja saadab päringu panka.

CORSi eesmärk on enne kontrollida, kas päringu alustava veebilehe domeeninimi on sama mis on päringu sihtkoha domeeninimi. Selleks küsib veebilehitseja enne pangalt millised domeenid on lubatud (Access-Control-Allow-Origin: swedbank.ee) ja kuna Charlie kodulehel on teine domeen (myblog.com) siis veebilehitseja katkestab päringu ning hoiatab veateatega “Cross Origin Request Blocked: The same origin policy does not allow reading of remote resources at https://swedbank.ee/transactions”

CORS töötab ainult veebilehitsejates. Selle eesmärk ei ole takistada inimestel avalikkusele kättesaadavate andmete saamist. Kui Charlie saab Bobi panga authorization tokeni kusagilt mujalt siis CORS seda ei takista.

Kuidas vältida CORS-põhiseid rünnakuid:

- Nõuetekohane konfiguratsioon

Kui veebiressurss sisaldb tundlikku teavet, tuleks päritolu nõuetekohaselt määrata Access-Control-Allow-Origin päises.

- Lubada ainult usaldusväärseid saite (enda loodud või partnerite saidid)

Access-Control-Allow-Origin päises määratud päritolu peaks olema ainult usaldusväärsed saidid.

- Vältige päise "Access-Control-Allow-Origin: null" kasutamist.

## CSRF

CSRF (Cross-site request forgery) on veebi turvaauk, mis võimaldab ründajal sundida kasutajaid sooritama tegevusi, mida nad ise ei kavatse teha. Eduka CSRF-rünnaku korral paneb ründaja ohvri kasutaja tahtmatult sooritama toiminguid nagu parooli vahetamine, aadressi muutmine või rahaülekande tegemine.

Kõige jõulisem viis CSRF-rünnakute vastu kaitsmiseks on lisada CSRF-tokeni päringutele. Token peaks olema:

- Ettearvamatu (suur juhuslik väärthus, mis on genereeritud turvalise meetodi abil)
- Seotud kasutaja seansiga
- Lisatud enne toimingu sooritamist

## XSS

XSS (Cross-site scripting) on veebi turvaauk, mis võimaldab ründajal käivitada kasutaja veebilehitsejas pahavaralist JavaScripti. XSS toimib haavatava veebilehe manipuleerimise teel, nii et see tagastab kasutajatele pahatahtliku JavaScripti. Kui pahatahtlik kood täidetakse ohvri brauseris, saab ründaja täielikult ohustada nende suhtlemist rakendusega.

Näiteks võib tulla ette olukord kus häkker lisab sotsiaalmeedias oma tutvustuse teksti sisse HTML <script> elemendi koodiga mis püüab teha pangas ülekannet või suunata sind kolmandale lehele - takistades sellega sul lehekülje kasutamist. Kui koduleht ei suuda tuvastada, et kirjelduse sees on HTML element lisab ta selle lehele ja kõik külastajad kes häkkeli profiili vaatavad nende arvutis käivitub ründav JavaScript.

Ennetamiseks tuleks:

- Kontrollida sisendit, mida kasutaja on sisestanud - näiteks kas see sisaldb JavaScripti
- Kodeerida andmete väljundit, viia HTML sümbolid "<", ">" teisele kujule "&lt;" ja "&gt;" mis takistab selle elemendi käivitamist
- Kasutada õigeid vastuspealkirju (response headers)
- Kasutada sisuturbpoliitikat (CSP)

## SQL Injection

SQL-Injection on veebi turvaauk, mis võimaldab ründajal sekkuda päringutesse, mida rakendus teeb oma andmebaasile. Ründajal on võimalik vaadata teiste kasutajate andmeid või mis tahes muid andmeid, millele rakendus ise saab juurdepääsu. Paljudel juhtudel saab ründaja neid andmeid muuta või kustutada, põhjustades muudatusi rakenduse sisus või käitumises.

Enamikku SQL-Injection juhtumeid saab vältida, kasutades päringu sees stringide ühendamise asemel parameetriseeritud päringuid (parameterized queries)

## Forced browsing

Forced browsing on veebi turvaauk, mille põhjuseks on hooletu kodeerimine ja mis võimaldab ründajale juurdepääsu ressurssidele, millele ta ei tohiks ligi pääseda. Forced browsingut kasutatakse, et saada juurdepääs piiratud lehekülgedele või muudele tundlikele ressurssidele veebiserveris, sundides URL-i. Kui veebiserveris asuvatele piiratud URL-idele, skriptidele või failidele ei ole kehtestatud asjakohast autoriseerimist, võivad need olla haavatavad forced browsing rünnakute suhtes.

Selleks, et vältida Forced browsing rünnakuid, tuleks kasutada nõuetekohast juurdepääsukontrolli ja lubada ainult turvalisi URL-e läbi rakenduse.

### Kontrollküsimused

1. Kuidas saab rakendust kaitsta CSRF rünnakute eest?
2. Mis rünnak toimub läbi päringute?
3. Kas CSRF rünnakut on võimalik kaitsta CORS lubamisega?

## 3.7 Pentest vahendite kasutamine oma veebisaidi turvalisuses veendumiseks (nt WPScan, OWASP ZAP)

Pentest ehk *Penetration test* on simuleeritud küberrünnak arvutisüsteemi vastu, et kontrollida, kust on võimalik midagi katki teha. Pen-testimine võib hõlmata mis tahes arvu rakendussüsteemide (nt rakendusprotokolli liidesed (API'd), frontend/backend serverid) rikkumiskatseid, et avastada haavatavusi nagu analüüsimaata sisendid, mis on vastuvõetlikud koodisisestusrünnakutele.

Penttestimine järgib tavaliselt järgmisi etappe:

- Uurimine - testija püüab teada saada, mida testitav süsteem vajab. See hõlmab püüdlust teha kindlaks, milline tarkvara on kasutusel, millised lõpp-punktid on olemas, millised parandused on paigaldatud jne. See hõlmab ka veebilehel varjatud sisu, teadaolevate haavatavuste ja muude nõrkade kohtade otsimist.
- Rünnak - testija üritab teadaolevaid või oletatavaid haavatavusi ära kasutada, et tõestada nende olemasolu.

- Aruanne - testija annab aru oma testimise tulemustest, sealhulgas haavatavustest, nende kasutamise viisist ja raskusastmest ning kasutamise raskusastmest.

## WPScan

WPScan (WordPress security scanner) on avatud lähtekoodiga WordPress turvalisuse lugeja. Seda saab kasutada oma WordPressi veebisaidi skaneerimiseks teadaolevate haavatavuste leidmiseks. Kuna tegemist on WordPressi musta kasti (black box) lugejaga, siis see imiteerib reaalset ründajat. See tähendab, et see ei tugine testide läbiviimiseks mingisugusele juurdepääsule WordPressi armatuurlauale või lähtekoodile. Teisisõnu, kui WPScan suudab leida teie WordPressi veebisaidil haavatavuse, siis suudab seda teha ka päris ründaja.

## OWASP ZAP

OWASP Zed Attack Proxy (ZAP) on üks populaarsemaid tasuta turvavahendeid maailmas ja seda hooldab aktiivselt rahvusvaheline vabatahtlik meeskond. Selle abil on võimalik automaatselt leida oma veebirakendustes turvaauke, samal ajal arendades ja testides oma rakendusi. Samuti on see suurepärane vahend kogenud pentesteritele, mida nad saavad kasutada käsitsi turvatestimiseks.

### Kontrollküsimused

1. Miks on oluline kasutada Pentest vahendeid?
2. Kas teenused jäavad pentest ajal kätesaadavaks?
3. Kas Pentestimine kahjustab rünnakuid läbi tehes süsteemi?

ÕV4: Dokumenteerib loodavad ja olemasolevad liidesed (liidestatud süsteemid, integratsioonipunktid, integratsioonimeetodid, turvalisuse reeglid);

## 4. VEEBITEENUSTE DOKUMENTEERIMINE JA SELLE VAJALIKKUS KOLMANDATE OSAPPOOLTE JAOKS

*Seda teemat käsitletakse põhjalikumalt õppekavaüleses praktilises projektis.*

Dokumentatsiooni olemasolu aitab jälgida rakenduse köiki aspekte ja see parandab tarkvaratoote kvaliteeti. Dokumentatsiooni peamine rõhk on arenduse ja hoolduse teadmiste

edasiandmine teistele. Hästi kirjutatud dokumentatsioon muudab teabe kergesti kättesaadavaks, aitab uutel kasutajatel kiiresti õppida, lihtsustab toodet ja vähendab kulusid.

Veebiteenuse hõlmab peamiselt andmete jagamist ja hoidmist. Et veebiteenust kasutada, peab kõigepärast uurima mida teenus pakkuda suudab. Dokumentatsioon annab selle kohta ülevaate ning täpsustab kõik võimalused ära. Hea dokumentatsioon õpetab tehniliselt kuidas teenusega ühendust luua ja päring kokku panna.

*„You can have the best, functional product, but no one will use it if they don't know how to.”*

Vaatamata standardite olemasolule on enamik veebilahendusi loodud rätsepatööna organisatsioonide sees ning need pole algsest mõeldud koostööks teiste süsteemidega. Klientide ja partnerite paremaks teenindamiseks on erinevad teenused siiski vaja koos tööle panna.

Traditsioonilised integratsioonimeetodid on mõeldud selliste väljakutsete lahendamiseks. Paraku toovad need kaasa mitmeid muid väljakutseid:

Ajakulu - muudatusi on tänapäeval vaja teha kiiresti. Kuigi integratsioonimeetodid on standardsed, tuleb liidestused kõikidesse osasüsteemidesse teha ikkagi rätsepatööna. See eeldab tavaliisi arendusmeetodeid, põhjalikku testimist jne. Kiired muudatused on võimalikud ainult siis, kui integreeritavad osad on ühe omaniku valduses. Erinevate omanike süsteemide integratsioon on oluliselt aeganõudvam.

Ebaratsionaalne - isegi kui osapooled soovivad koostööd teha, võib integratsioon osutuda nii keeruliseks, isegi võimatuks, et sellesuunalised pingutused muutuvad ebamõistlikuks. See aga tähendab teiselt poolt möödalastud ärvõimalusi ja suuremaid ärikulusid.

Integratsionikulud - integratsioonitöödeks võib vaja minna mahukaid arendustöid kõrgekvalifikatsiooniga spetsialistide poolt, mis tähendab väga suuri investeeringuid.

## 4.1 OpenAPI spetsifikatsioon

Swagger nime all on tööriistad, mis aitavad teil kujundada, dokumenteerida ja testida oma API'sid mastaapselt.

Swaggeri kasutajaliides võimaldab kasutajatel visualiseerida ja suhelda API ressurssidega, ilma et neil oleks olemas rakendusloogika. See genereeritakse automaatselt teie OpenAPI spetsifikatsioonist, kusjuures visuaalne dokumentatsioon muudab lihtsaks tagasiside rakendamise ja kliendipoolse tarbimise.

**Swagger Petstore** 1.0.6

[ Base URL: [petstore.swagger.io/v2](http://petstore.swagger.io/v2) ]  
<https://petstore.swagger.io/v2/swagger.json>

This is a sample server Petstore server. You can find out more about Swagger at <http://swagger.io> or on [irc.freenode.net #swagger](#). For this sample, you can use the api key `special-key` to test the authorization filters.

Terms of service  
Contact the developer  
Apache 2.0  
Find out more about Swagger

Schemes

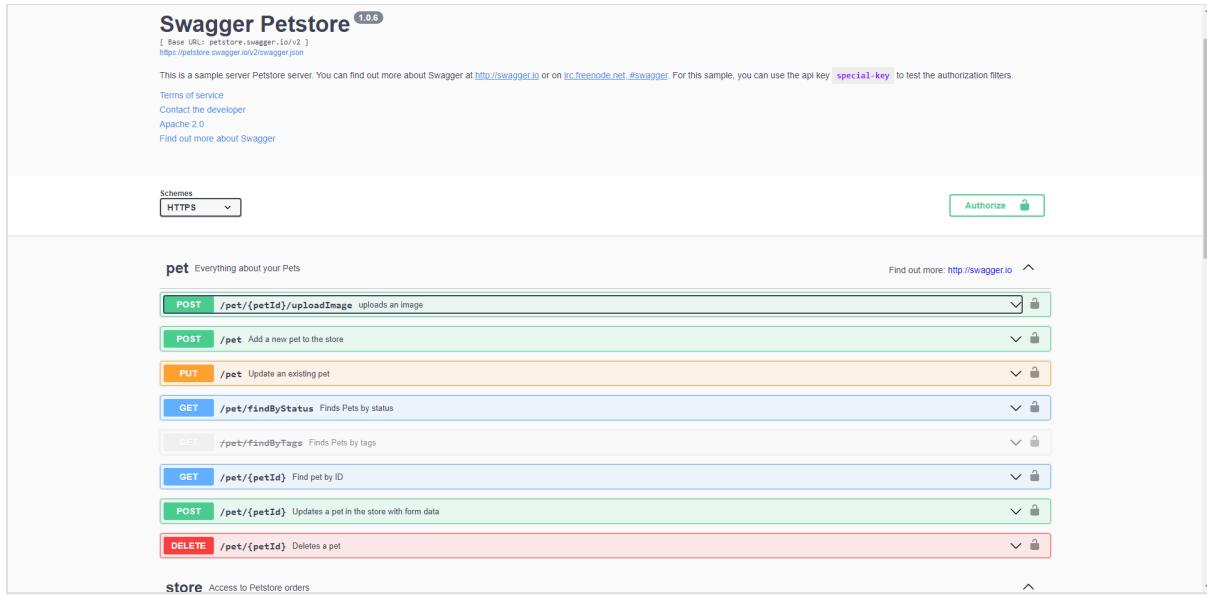
**pet** Everything about your Pets

Find out more: <http://swagger.io> ^

<b>POST</b>	<a href="#">/pet/{petId}/uploadImage</a> uploads an image	<input checked="" type="checkbox"/>
<b>POST</b>	<a href="#">/pet</a> Add a new pet to the store	<input type="checkbox"/>
<b>PUT</b>	<a href="#">/pet</a> Update an existing pet	<input type="checkbox"/>
<b>GET</b>	<a href="#">/pet/findByStatus</a> Finds Pets by status	<input type="checkbox"/>
<b>GET</b>	<a href="#">/pet/findByTags</a> Finds Pets by tags	<input type="checkbox"/>
<b>GET</b>	<a href="#">/pet/{petId}</a> Find pet by ID	<input type="checkbox"/>
<b>POST</b>	<a href="#">/pet/{petId}</a> Updates a pet in the store with form data	<input type="checkbox"/>
<b>DELETE</b>	<a href="#">/pet/{petId}</a> Deletes a pet	<input type="checkbox"/>

**store** Access to Petstore orders

^



## Pilt: Swagger UI demo