Oskari Oksanen

Web-sivujen teko Node.js:llä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

22.5.2017

Tekijä(t) Otsikko Sivumäärä Aika	Oskari Oksanen Web-sivujen teko Node.js:llä 34 sivua 22.5.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Yliopettaja Erja Nikunen

Insinöörityön aiheena oli perehtyminen JavaScript-pohjaiseen Node.js-web-palvelimeen ja erilaisten sille olevien kehys- ja sivupohjavaihtoehtojen vertailu keskenään. Vertailujen lisäksi perehtyminen sisälsi tutustumista erilaisiin Node.js -ohjelmointikäytäntöihin, ja Node.js:n taustaan. Innoittaja aiheeseen oli samaan aikaan Node.js:llä tekemäni ohjelmistoprojekti.

Lopputulos ei kata aivan kaikkea, ja esimerkiksi tietokantakerroksesta en kirjoittanut ollenkaan, koska työssä lähtökohtana oli rakentaa hyvin staattinen sivusto. Kehyskirjastoja vertailin keskenään neljää, mutta niitäkin olisi ollut runsaasti enemmän joista valita. Kuitenkin olen tyytyväinen lopputulokseen, ja tuntuu kuin olisin käynyt relevanteimmat osat Node.js:stä läpi, ja sisäistänyt alustan hyvin.

Avainsanat	Node.js, JavaScript, Express.js, Hapi, Koa.js, Sails.js, Pug, Handlebars, React, Istanbul, Grunt, web-sovellus, sivupohja, kehys

Author(s) Title Number of Pages Date	Oskari Oksanen Construction of Web Software Using Node.js 34 pages 22 May 2017
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	Software Engineering
Instructor(s)	Erja Nikunen, Principal Lecturer

The purpose of this Bachelor’s Thesis was to study the JavaScript based Node.js runtime environment. The focus of the thesis is evaluating the existing web framework libraries and templating engines for the platform, and studying the best practices but the thesis also includes delving into the history and the background of Node.js.

The resulting document misses some things, such as the database layer, which was deemed mostly unnecessary because the product to be made was mostly static. There could have also been more libraries included in the evaluation phase: for example, the framework evaluation focused only on the four biggest web frameworks. However, the study met the objectives set for it and provided the author with a lot of information and knowledge of Node.js.

Keywords	Node.js, JavaScript, Express.js, Hapi, Koa.js, Sails.js, Pug, Handlebars, React, Istanbul, Grunt, web software

Sisällys

Lyhenteet

1 Johdanto 1

2 Sovellusympäristö 1

2.1 JavaScript 1
2.2 Node.js:n lyhyt historia 2
2.3 Yleistä Node.js:stä 2
2.3.1 Tapahtumasilmukka 4
2.3.2 Lupaus 5
2.3.3 Prototyyppiperinnöllisyys 6

3 Kehykset 7

3.1 Määritelmä 7
3.2 Vertailu 8
3.3 Express-kehys 10
3.4 Hapi-kehys 12
3.5 Koa.js-kehys 15
3.6 Sails.js-kehys 17
3.7 Päätelmät 18

4 Sivupohjat 19

4.1 Pug-sivupohja 20
4.2 Handlebars-sivupohja 21
4.3 React-sivupohja 22

5 Toteutus 23

5.1 Kansiorakenne 23
5.2 Riippuvuudet 24
5.3 Reititys 25
5.4 Sivupohjat 26
5.5 Omien kirjastojen luonti 26
5.6 Testaus ja laadunvarmistus 27
5.6.1 Yksikkötestaus 27

5.6.2 Koodin tyylin tarkistus 28
5.6.3 Kattavuuden testaus Istanbulilla 29
5.6.4 Grunt-automatisaatio 30

6 Yhteenveto 32

Lähteet 33

Lyhenteet

AJAX	Asynchronous JavaScript And XML on tapa, jolla web-sivu voi tehdä palvelinpyyntöjä tausta-ajona ilman, että koko sivua tarvitsee ladata uudestaan.
API	Application programming interface on rajapinta, jota käyttämällä ulkopuolinen sovellus voi käyttää kyseisen esim. web-sivun tai kirjaston tai palveluja.
BDD	Behavior-driven development on yksikkötestauksen yksi alatyyli.
CSS	Cascading Style Sheets on tyyliohje, jolla voi määritellä web-sivun ulkonäön.
HTTP	Hypertext Transfer Protocol on protokolla, jota selaimet ja web-palvelimet käyttävät tiedonsiirtoon.
JSON	JavaScript Object Notation on tapa hahmottaa tietoa.
MVC	Model–view–controller on tapa hahmottaa ohjelmisto, joka painottaa, että näkymä (view), käsittelijä (controller) ja tieto (model) pitää olla erillään.
NPM	Node.js Package Manager on Node.js:n mukana tuleva kirjastojenhallintatyökalu.
TDD	Test-driven development on yksikkötestauksen yksi alatyyli.
XML	Extensible Markup Language on tapa hahmottaa tietoa.

1 Johdanto

Opinnäytetyön tavoitteena on tutkia Node.js:ää ja JavaScriptiä läpi yleisesti ja vertailla erilaisia Node.js:n kehys- sekä sivupohjakirjastoja. Lisäksi tarkoituksena on käydä jonkin verran yleisiä käytäntöjä läpi, mitä sivua rakentaessa voi tarvita.

Kiinnostuin aiheesta alun perin alkutalvesta 2015, kun värväydyin auttamaan erästä internetyhteisöä heidän kotisivujensa ylläpidossa. Pian kävi ilmi, että kotisivut tarvitsivat paljon uusimista, ja päätimme yhdessä, että Node.js voisi olla hyvä alusta uusille sivuille, koska JavaScript oli kaikille tuttua. Kyseinen internetyhteisö kuitenkin sulki ovensa hieman vuotta myöhemmin, eivätkä projektia varten tekemäni sivut koskaan nähneet kunnollista käyttöä.

Lähtökohtana vertailuille on, että rakennettava web-sivu on hyvin staattinen, eikä sisällä dynaamista sisältöä. Käytän eri kehysten ja sivupohjien arvioinnissa hyväkseni omia kokemuksiani projektin kanssa, mutta myös lähdeviitattua tietoa. Toteutus-osion yleisten käytäntöjen suositukset pohjautuvat pääosin omiin kokemuksiini.

2 Sovellusympäristö

2.1 JavaScript

JavaScript on dynaaminen komentosarjakieli, jonka tarkoituksena on mahdollistaa web-sivujen dynaamisuus. Kieli näki ensimmäistä kertaa päivänvaloa Netscape Communications Corporationin kehittämässä NetScape-selaimessa vuonna 1996.

Nykyään JavaScriptia käytetään synonyyminä ECMAScript-standardille, joka määrittelee JavasScriptin ominaisuudet. Itse toteutus on jätetty mm. selainvalmistajien harteille. Totetutuksista, toisin sanottuna ajoympäristöistä, kenties tunnetuin on Googlen V8, joka toimii moottorina mm. Googlen omassa Chrome-selaimessa ja Node.js web-palvelimessa.

2.2 Node.js:n lyhyt historia

Node.js syntyi vuonna 2009 Ryan Dahlin halusta luoda tapahtumapohjainen palvelin. Teoriana toteutukselle oli, että palvelin vastaisi annettuihin pyyntöihin asynkronisesti [1], ja mahdollistaisi täten minimaaliset vasteajat verrattuna synkronisesti toimiviin verrokkeihinsa. Dahl harkitsi monia kieliä palvelintyyppinsä toteuttamiseen kuten C:tä ja Haskellia, mutta päätyi JavaScriptiin Googlen julkistaessa samoihin aikoihin uuden V8 JavaScriptin-ajoympäristön. Dahl on kertonut, että päätöstä käyttää kielenä JavaScriptiä tuki myös, ettei sille ollut runsaasti vakiintuneita kirjastoja, ja asynkronisuuden pakottaminen oli täten suhteellisen helppoa.

Pian Node.js:n syntymän jälkeen pilvipalvelinyhtiö Joyent palkkasi Dahlin ja osti kaikki oikeudet Node.js:n. Dahl jatkoi kaupan jälkeen Node.js:n pääkehittäjänä aina vuoteen 2012, jolloin hän vetäytyi julkisuudesta.

Dahlin lähdettyä vuonna 2012 Node.js:n pääkehittäjäksi nousi Node.js:n paketinhallintajärjestelmän kehittäjä Isaac Schlueter. Schlueter kuitenkin erkani projektista sisäisten erimielisyyksien vuoksi ja perusti usean muun avainkehittäjän kanssa Node.js:n kanssa samasta aivopääomasta kilpailevan Io.js:n. Nykyään Node.js:n koodikannan hallinnoinnista ja jatkokehittämisestä vastaa Joeynt [2].

2.3 Yleistä Node.js:stä

Käynnistyessään Node.js luo palvelimen, joka kykenee vastaamaan palvelupyyntöihin sekä jakamaan staattisia tiedostoja, josta johtuen se ei välttämättä tarvitse erillistä palvelinkerrosta, kuten Apachea tai Nginxiä, reitittämiseen. Node.js toimii yhden säikeen varassa, joten se käyttää oletuksena vain yhden prosessoriytimen. Node.js:n kuitenkin saa käyttämään useampaa prosessoriydintä luomalla useita prosesseja, jotka voivat olla joko Node-palvelimen lapsiprosesseja tai erillisiä palvelimia [3].

Päinvastoin kuin saattaisi luulla, Node.js on nopea yksisäikeisyytensä ansiosta. Node.js kykenee käsittelemään keskimäärin 8000 palvelupyyntöä sekunnissa [4]. Nopeudesta voi kiittää Node.js:n tapahtumasilmukkaa (ks. 2.3.1), jonka ansiosta palvelin kykenee vastaanottamaan enemmän pyyntöjä vähemmällä resurssimäärällä kuin vastaava oman säikeen jokaiselle pyynnölle tekevä sovellus [5]. Node.js:n nopeudelle on kuitenkin olen-

naista, että palvelinpyynnöt ovat nopeita toteuttaa, sillä levyoperaatiot ovat hitaita ja paljon prosessoriaikaa vaativat tehtävät estävät tehtävän aikana tulevien pyyntöjen huomioimisen [6]. Nämä rajoitukset voidaan kuitenkin tarvittaessa kiertää suorittamalla ne muualla kuin palvelupyyntöjä vastaanottavalla palvelimella.

Node.js:n tapahtumapohjaisuudella on kuitenkin myös kääntöpuolensa: sovellusympäristö on luonteeltaan asynkroninen, ja palauttaakseen sen osittain synkroniseksi kehittäjien on täytynyt käyttää takaisinkutsuja. ”Callback hell”, eli takaisinkutsuista johtuva koodin puuroutuminen (koodiesimerkki Koodiesimerkki 1), on asia, mistä usein valitetaan Node.js:n teknisiä puolia ruotiessa [7]. Yleensä selvä merkki puuroutumisesta on että ohjelman rakenne on vaikeasti hahmotettavissa johtuen useammasta etenemisreitistä. Puuroutumiseen yleensä liittyy myös ns. ”tuomion pyramidi”, joka kohoaa sivun oikeaa laitaa kohden sisennyksien määrän kasvaessa.

```
var fs = require('fs');

function readJSON(filename, callback) {
  fs.readFile(filename, 'utf8', function(err, file) {
    if(err) {
      return callback(err);
    }
    try {
      parsedJSON = JSON.parse(file);
    } catch (ex) {
      return callback(ex);
    }
    callback(null, parsedJSON);
  });
}

var name = 'package.json';
readJSON(name, function (err, json) {
  if (err) {
    console.log('error happened: ' +);
  } else {
    console.log(json);
  }
})
```

Koodiesimerkki 1. Useista takaisinkutsuista johtuva koodin puuroutuminen havainnollistettuna.

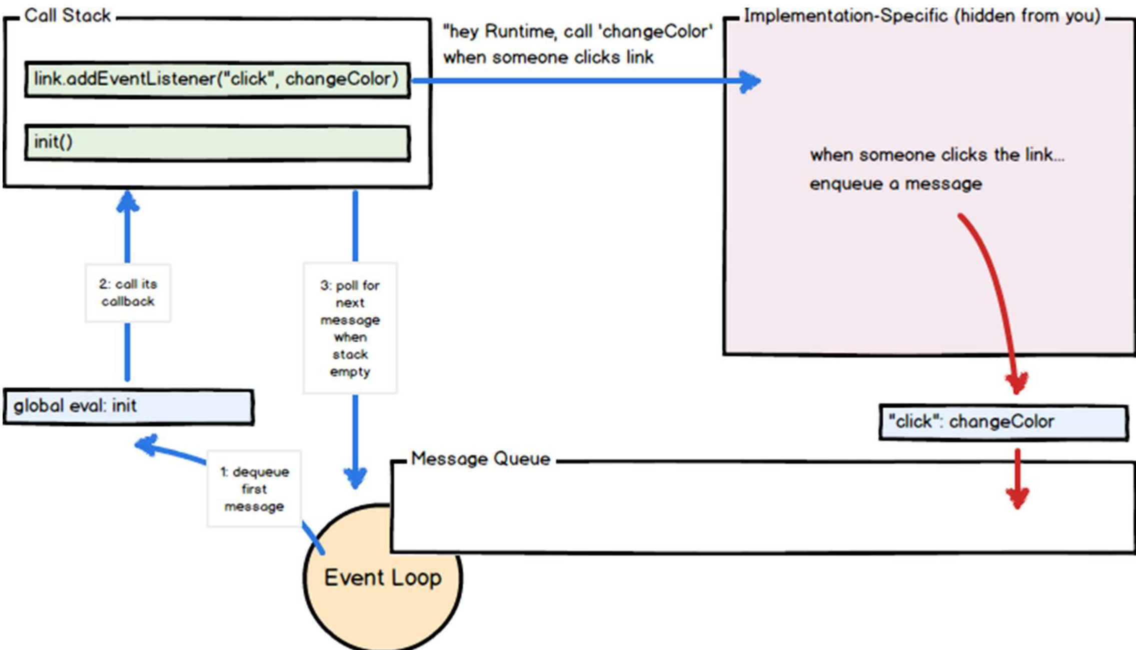
Koodinpuuroutumisen lisäksi Node.js:ää vaivaa samat asiat kuin Javascriptiä yleensä, kuten siirtyminen prototyyppiperinnöllisyyteen (ks. 2.3.3) perinteisten oliokielien perinnöllisyydestä sekä yleinen kypsymättömyys. Kuitenkin uusi ECMAScript 6 -standardi on tuonut sovellusympäristöön parannuksia, joista esimerkkeinä mm. aiemmin vain kolman-

nen osapuolen kirjastoista löytyvät lupaukset (ks. 2.3.2) sekä class-avainsanan. Lupauksilla voidaan tietyissä määrin ehkäistä koodin puuroutumista, kun taas class-avainsana piilottaa prototyyppiperinnöllisyyden konepellin alle.

Kirjastojenhallintaa varten Node.js:llä on oma järjestelmänsä, NPM. NPM on suhteellisen helppokäyttöinen, vaikka kärsiikin yleisesti paketinhallintajärjestelmiä vaivaavasta viasta, eli ettei kirjastoja ole helppoa verrata keskenään hankaloittaen tehtävään parhaan mahdollisen kirjaston valintaa [7].

2.3.1 Tapahtumasilmukka

Node.js on tapahtumapohjainen, eli se käyttää tarkkailija-suunnittelumallia. Suunnittelumallissa sitä noudattava osapuoli on passiivinen, kunnes saa ärsykkeen; Node.js:ssä tapahtumasilmukka (event loop, kuva Kuva 1) käy tapahtumajonoansa läpi, ja reagoi kuhunkin ohjeistetulla tavalla. Jos tapahtuma tarvitsee jatkotoimia ohjeistuksen jälkeen, menee toimi tapahtumajonon perälle, josta se sitten suoritetaan vuoron tullessa kohdille.



Kuva 1. Node.js:n tapahtumasilmukka [8].

Tapahtumasilmukka on asynkroninen, eikä asioiden tapahtumisjärjestys ole taattu: Koodiesimerkki Koodiesimerkki 2:n mukaisen ohjelman tulostama palvelupyynnöstä saatu vastaus olisi todennäköisesti tyhjä, koska Node.js ei jää odottamaan vastausta pyyntöön, vaan pyynnön lähettämisen jälkeen siirtyy suoraan tulostuskomentoon. Yksi tapa

saada suoritustapa taas synkroniseksi olisi ollut asettaa tulostuskomento takaisinkutsu-funktion (callback) sisään, niin että Node.js suorittaa sen vasta saatuaan palvelupyyntöön vastauksen.

```
var html = '';
request('http://www.google.com', function(error, response, body) {
 html = body;
});
console.log('Saatu vastaus: ' + html);
```

Koodiesimerkki 2. Node.js tekee palvelinpyynnön, mutta tulostaa vastauksen ennenaikaisesti.

2.3.2 Lupaus

Yksinkertaisimmillaan lupaus (promise) tarkoittaa väliaikaista muuttujaa, joka korvaa oikeaa arvoa, joka ei ole heti saatavissa. Suoritusympäristö jättää lupauksen odottamaan, ja palaa lupauksen erääntyessä, eli silloin kun muuttuja saa varsinaisen arvonsa, suorittamaan muuttujasta riippuviksi määritetyt jatkotoimenpiteet; seurauksena tästä halutut koodilohkot suoritetaan keskenään synkronisesti.

JavaScriptiin lupaukset ovat tulleet ECMAScript 6 -määrityksen myötä. Kuitenkaan Node.js:n tarjoamat perusfunktiot eivät sellaisinaan ole käytettävissä lupauksien kanssa, vaan yleensä niistä täytyy erikseen tehdä lupauksia käyttävä muunnos (koodiesimerkki Koodiesimerkki 3), lisäten jonkin verran rutiinikoodin kirjoittamisen tarvetta (boilerplate code).

```
var fs = require('fs');

// tehdään Node.js:n vakiokirjaston readFile-funktiosta
// lupauksien käyttöä tukeva versio
function readFileP(filename){
  return new Promise(function(resolve, reject){
    fs.readFile(filename, 'utf8', function (err, file){
      if(err) {
        reject(err);
      } else {
        resolve(file);
      }
    });
  });
}

// tuloksena saadaan koodiesimerkki 1:n vastaavaa
// funktiotalukukelpoisempi readJSON-funktio
function readJSON2(filename){
  return readFileP(filename).then(function (file) {
    return JSON.parse(file);
  });
```

```
}

//käyttöesimerkki
readJSON2('package.json')
 .then(console.log) // console.log on funktio joten voidaan lyhentää
 .catch(function (err) (
   console.log('error happened: ' + err)
 ));
```

Koodiesimerkki 3. Koodiesimerkki 1 käyttäen lupauksia.

Natiivitoteutuksen lisäksi Node.js:lle on myös olemassa useita kirjastoja, jotka helpottavat lupauksien käyttöönottoa tarjoamalla ylimääräistä syntaktista sokeria: esimerkiksi Bluebird-kirjasto tarjoaa `promisify`-apufunktion, jolla takaisinkutsumallia noudattavat funktiot voidaan muuttaa käyttämään lupauksia yhdellä funktiokutsulla vähentäen rutiinikoodin kirjoittamisen tarvetta siirtyessä ohjelmointimallista toiseen.

2.3.3 Prototyyppiperinnöllisyys

Usea oliopohjaisista kielistä tuleva ohjelmoija kompastuu JavaScriptin perinnöllisyyteen. Kielissä kuten Javassa perinnöllisyys tapahtuu siten, että uutta oliota luodessa ensin luodaan alimmasta pohjaluokasta ilmentymä, jonka päälle sitten kasataan aliluokkien ominaisuuksia. JavaScriptin prototyyppiperinnöllisyys eroaa perinteisestä ajatusmallista siinä, että olio ei heti tiedä kaikkia perittyjä ominaisuuksiaan. JavaScript-olion ominaisuutta kutsuessa ajaja (runtime) etsii ensin haettua ominaisuutta olion omista ominaisuuksista. Jos ominaisuutta ei löydy, käy ajaja olion prototyyppipinoa läpi, kunnes haettu ominaisuus joko löytyy perityistä prototyypeistä tai peritty prototyyppipino on käyty läpi [9]. Käytännössä oliolla voi siis olla jokin ominaisuus moneen kertaan määriteltynä, mutta vain viimeisin määritelmä otetaan huomioon. Kuitenkin aiemman määritelmän voi saada esille erikseen kaivamalla prototyyppipinoa.

Kuten perinteisemmissä oliokielissä, on avainsana this tärkeä myös JavaScriptissä, jota käytetään viitatessa nykyiseen olioon. Toisin sanoen: jos kutsutaan perityn luokan funktiota, jonka sisällä käytetään avainsanaa this, suosii ajaja nykyisen olion ominaisuuksia, jotka yliajavat prototyypeistä perityt.

Jokainen saman aliluokan ilmentymä jakaa saman prototyyppipinon. Jos muokataan yhden aliluokan ilmentymän prototyyppijonoa, näkyy muutos myös jokaisessa muussa samaa pinoa ilmentävässä oliossa.

3 Kehykset

Osiossa on tarkoitus vertailla erilaisia Node.js:n kanssa käytettäviä taustakehyksiä. Tämä aloitetaan määrittelemällä ensin kehys (luku 4.1), jonka jälkeen edetään vertailtavan kehysjoukon rajaamiseen (luku 4.2). Kun kehysjoukko on selvä, käydään ne läpi yksitellen (luku 4.3 – 4.6). Vertailun lopuksi tehdään yhteenveto (luku 4.7).

3.1 Määritelmä

Node.js:stä puhuttaessa kehyksellä (framework) tarkoitetaan taustakehystä (back-end framework), joka on palvelimen, eli tässä tapauksessa Node.js:n, päälle tuleva kerros, joka sisältää pyynnönkäsittelylogiikan. Vertauskuvallisesti palvelin on eräänlaisen koneen ulkoiset funktiot, kuten sisäänotto- ja ulosantoluukku, ja kehys on, mikä määrittelee koneen sisälmysten pohjamallin.

Back-end-kehysten lisäksi on myös edustakehyksiä (front-end framework), kuten Angular, jotka eroavat taustakehyksistä siinä, että valtaosa näytettävän sivuston muodostamisesta suoritetaan käyttäjän selaimessa Nodej.js-palvelimen sijaan. Etuna tässä lähestymistavassa on, ettei kehittäjän tarvitse pitää mielessä Node.js:n asynkronisuutta. Edustakehykset sisältävät yleensä työkalut sivupohjien muodostamiseen, ja reitityksen.

Yleisesti kehysten ominaisuudet vaihtelevat suuresti: On mikrokehyksiä, jotka sellaisenaan tarjoavat vain kaikkein välttämättömimmän, ja antavat kehittäjille verrattain vapaat kädet täydentää kehystä valitsemillaan ominaisuuksilla, kuten Express; olemassa on myös täyspinokehyksiä (full-stack framework), jotka tarjoavat tuen käyttöliittymän toteutuksesta aina tietokantatallennukseen asti, kuten Meteor, ja syvästi erikoistuneita kehyksiä, kuten Restify. Välttämättömimpiin asioihin, joita kehykset tarjoavat, voidaan lukea reitityksen määrittely, pyyntöjen käsittely väliohjelmien (middleware) avulla, sekä yleinen Node.js:n syntaktisella sokerilla kuorruttaminen.

Projektissa käytettävää kehystä valitessa tulisi olla tietoinen omista tarpeistaan. Jos projekti aloittaa tyhjältä pöydältä tai käyttää jo valmiiksi joitain osia yleisemmistä Node.js-kirjastopinoista, kuten MongoDB:tä tai Angular.js:ä, voi kaiken kattava kehys kuten Meteor olla omaan tarpeeseen sopiva. Muutoin voi olla parempi valita jokin minimalistisempi kehys, jonka voi räätälöidä omiin tarkoituksiin sopivaksi, kuten Express.

3.2 Vertailu

Node.js-projekti tarvitsee kehyksen, joten niiden keskinäinen vertailu on aiheellinen. Ennen kuin kehyksiä voi kunnolla vertailla keskenään, tulee tiedostaa niiden kattavuuden erilaisuus. Esimerkiksi sivusto nodeframework.com jakaa kehykset viiteen kategoriaan:

1) Sinatramaiset: mikrokehykset jotka tarjoavat monipuoliset työkalut ja antavat vapaat kädet muokata kehystä haluamaansa suuntaan.

2) Railsmaiset: mikrokehykset jotka tarjoavat projektipohjan (template) jonka päälle rakentaa.

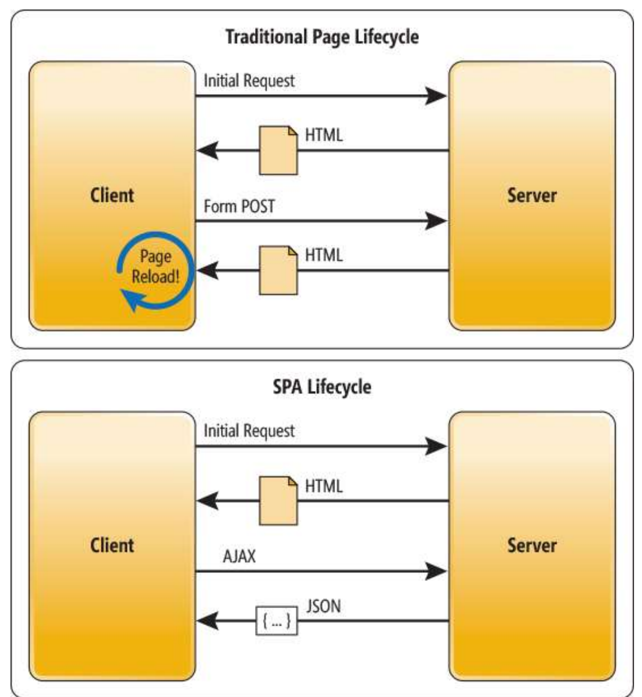
3) Täyspinokehykset: nimensä mukaisesti kehykset jotka tarjoavat työkalut niin back-end kuin edusta -kehitykseenkin.

4) REST API -kehykset: kehykset nopeaan REST API:n tarjoavan palvelimen luontiin.

5) Muut kirjastot.

Kategorisointi helpottaa projektiin sopivan kehyksen valintaa, ja voin esimerkiksi karsia kaikki REST API -kehykset sekä muut kirjastot omasta kehysvertailustani sen perusteella, etteivät ne täytä projektini vaatimuksia.

Täyspinokehyksien vahvuus on tapahtuneiden muutosten näyttäminen reaaliaikaisesti käyttäjille, sekä ns. yhden sivun sovellukset (SPA). SPA-sivustot eroavat muista siinä, että ne hakevat koko sivuston kerralla palvelimelta, ja ”siirtymät” eri osioiden välillä tehdään JavaScriptin avulla tehtävällä HTML-manipulaatiolla. Jos ensimmäisen latauksen jälkeen ilmenee tarvetta päivittää sisältöä, hoidetaan se AJAX:n kaltaisella asynkronisella pyynnöllä (kuva Kuva 2). Projektini valintavertailusta täyspinokehykset voidaan karsia pois, sillä valmisteltava sivusto on suurilta osin staattinen, eikä harvakseltaan päivittyvää uutissyötettä lukuun ottamatta juuri sisällä dynaamisia osia.

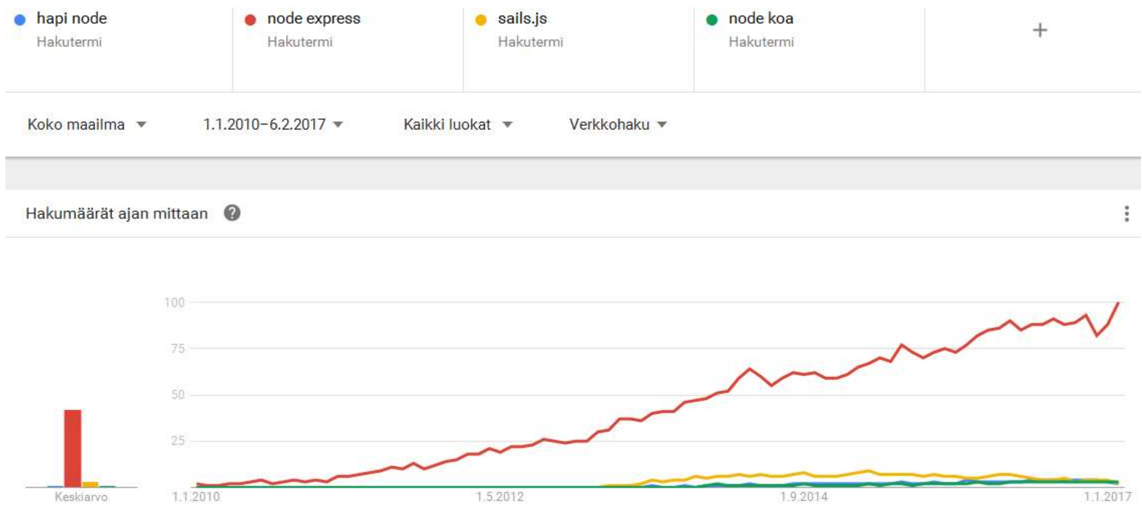


Kuva 2. SPA-sivun toimintalogiikka verrattuna perinteiseen web-sivuun [10].

Alkukarsinnan jälkeen muita kriteereitä kehyksen valintaan ovat yleisen suosion määrä sekä koodaajaystävällisyys. Kumpikin kriteeri on tärkeä ja vaikuttaa kehityksen tehokkuuteen. Suosio kertoo, kuinka helppoa ongelmakohdissa on saada tukea ja kuinka paljon alustalle on valmiita apukirjastoja, kun taas koodaajaystävällisyys on tekijänä siinä, kuinka selkeää koodia kehittäjä saa aikaiseksi. Koodaajaystävällisyys riippuu paljolti kehittäjän omasta mausta, mutta suosiota on helppo mitata vertaamalla GitHub-tähtien sekä Google Trends -osumien määrää.

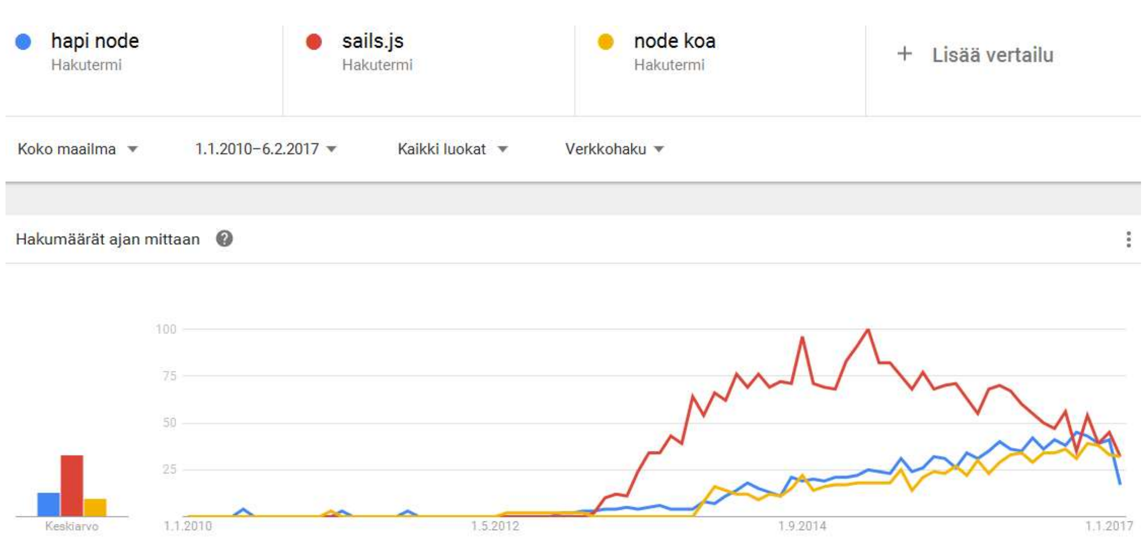
Yksi sivusto, jota voi käyttää vertailussa on suosituimpia Node.js-kehyksiä listaava Node Frameworks -sivusto. Vertailemalla sivustolla olevia kehyksiä kunkin GitHub-repositorion tähtimäärän mukaan ylivoimaisesti suosituimmaksi osoittautuu Express.js, harvojen muiden päästessä edes neljäsosaan sen tähtimäärästä. Seuraavassa on joitain GitHub-tähtimääriltään huomattavia kehyksiä (tilanne 6.2.2017):

- Express (29 812 tähteä)
- Sails.js (16 432 tähteä)
- Koa.js (13 671 tähteä)
- Hapi (7 357 tähteä)



Kuva 3. Hapi, Express, Sails.js ja Koa.js vertailtuna Google Trendsissä [11].

Vertaillessa neljää kehystä Google Trendsissä on viimeistään selvää, että Express on neljästä kaikkein suosituin (kuva Kuva 3). Kolmen muun vertailu keskenään on huomattavasti tasaväkisempää: Sails.js on kolmesta suosituin Koa.js:n ja Hapin ollessa lähes yhtä suosittuja (kuva Kuva 4).



Kuva 4. Hapi, Sails.js ja Koa.js vertailtuna Google Trendsissä [12].

3.3 Express-kehys

Express on yksi vanhimmista kehyksistä Node.js:lle, ja sen kehitys alkoi vuonna 2009 TJ Holowaychukkin toimesta. Aluksi Expressiä kehitettiin Connect-kehyksen päälle, mutta version 4 myötä Express on ollut itsenäinen. Suurin osa Connect-kirjastoista toimii

myös Expressissä, näin myös versiossa 4. Nykyään Expressin pääkehittäjä on IBM:n alainen StrongLoop, josta tuli kehyksen pääkehittäjä vuonna 2014 [13].

Expressiä voi pitää Noden de facto -kehyksenä, sillä sille on kaikista Node-kehyksistä eniten väliohjelmakirjastoja, ja se on kehyksistä suosituin (kuva Kuva 3). Expressiin on olemassa projektigeneraattori (express-generator-kirjasto), jolla voi luoda projektille kansiorakenteen sekä joitain perusasioita sisältäviä tiedostoja, kuten reititys- ja HTTP-palvelimen käynnistysmoduulin sekä projektin asetustiedoston.

```
var express = require('express');
var app = express();

// Logging middleware
app.use(function(request, response, next) {
  console.log("In comes a " + request.method + " to " + request.url);
  next();
});

// Send "hello world"
app.get('/', function (request, response) {
  res.send('Hello, World!');
});

var server = app.listen(3000, function() {
  console.log('Express is listening to http://localhost:3000');
});
```

Koodiesimerkki 4. Yksinkertaisen HTTP-palvelimen luonti käyttäen Expressiä.

Palvelinta luotaessa Expressillä otetaan ensin app-muuttujaan talteen instanssi express-moduulista, ja lopuksi kutsutaan app-muuttujan metodia listen (koodiesimerkki Koodiesimerkki 4), joka on käytännössä sama metodi kuin Noden http-moduulin createServer.

Väliohjelmia Express-palvelimelle voidaan lisätä antamalla app.use-metodille väliohjelman sisältävä funktio, joka lopuksi kutsuu kolmantena parametrina saatavaa takaisinkutsufunktiota next. Reititystä varten Expressissä on jokaiselle HTTP-metodille (GET, POST, jne.) oma metodinsa: reitittäessä metodille annetaan polku sekä takaisinkutsufunktio. Metodi ja polku yhdessä karsivat, millaisen pyynnön kohdalla takaisinkutsufunktiota kutsutaan.

Palvelupyynnön tullessa Express käy asetettuja väliohjelmia sekä HTTP-metodeja järjestyksessä läpi, kunnes joko löytyy response-objektin lähettävä funktio tai saavutaan funktiopinon loppuun.

```
var express = require('express');
var router = express.Router();

module.exports = router;

router.get('/', function (req, res) {
  res.send('Welcome to my platypus page.');
});
```

Koodiesimerkki 5. Kuvitteellisen platypus-sivun reititystiedosto platypus.js.

Modulaarisuutta Express-palvelimeen saadaan express.Router-luokan avulla: Ensin reitityksen osan sisältävässä moduulissa luodaan uusi Router, johon sitten lisätään uusia takaisinkutsufunktioita kuten normaalisti Expressissä (koodiesimerkki Koodiesimerkki 5). Lopuksi moduuli tuodaan pää-Express-ohjelmaan ja asetetaan kuten väliohjelma käyttäen app.use-metodia (koodiesimerkki Koodiesimerkki 6).

```
var platypus = require('./platypus');

...
app.use('/platypuses', platypus);
```

Koodiesimerkki 6. Koodiesimerkki 5:n reititystiedoston lisäys jo olemassa olevaan Express-palvelimeen.

3.4 Hapi-kehys

Hapi alkoi nousta yleiseen tietoon vuoden 2012 lopussa [14], mutta suosiota projekti on alkanut kerätä vasta vuodesta 2014. Vuoden 2017 alkutalvesta Hapi oli suosiossa samalla viivalla Koa.js:n kanssa häviten kuitenkin Sails.js:lle ja Expressille (kuva Kuva 4). Hapin pääkehittäjä on kautta projektin ollut Eren Hammer.

Projektin alkuvaiheessa tavoitteena oli luoda jo olemassa olevan kehyksen (Express) päälle tarpeet täyttävä väliohjelmisto. Hapin kehityksen edistyessä Hammerille kuitenkin kävi selväksi, ettei Expressin ohjelmistoarkkitehtuuri sovi hänen kaavailemalleen ohjelmistolle, ja hän päätti tehdä Hapista itsenäisen kehyksen. Teesejä Hapille Hammer asetti kaksi: Konfiguraatio on parempi kuin koodi ja bisneslogiikka tulee erottaa tiedonkuljetuskerroksesta [14].

Hapilla palvelimen luonti aloitetaan luomalla uusi server-luokan ilmentymä. Luonnin jälkeen luokkaan tehdään tarvittavat säädöt, ja lopuksi se käynnistetään start-metodilla.

Väliohjelmia Hapissa lisätään server-luokan ext-metodilla (koodiesimerkki Koodiesimerkki 7). Reititysfunktiot määritellään käyttäen server.route-metodia: metodille annettavista parametreistä tärkeimmät ovat HTTP-verbi (method), polku (path) ja käsittelijä (handler). Pyyntövastaus luodaan kutsumalla käsittelijän saamaa reply-funktiota.

```
var Hapi = require('hapi');
var server = new Hapi.server()
server.connection({ port: 3000 });

// do logging on each request
server.ext('onRequest', function (request, reply) {
  console.log("In comes " + request.method + " to " + request.path);
  return reply.continue();
});

// send 'hello world'
server.route({
  method: 'GET',
  path: '/',
  handler: function (request, reply) {
    reply('Hello, world!');
  }
});

server.start(function () {
  console.log('Hapi is listening to: ', server.info.uri);
});
```

Koodiesimerkki 7. Yksinkertaisen HTTP-palvelimen luonti Hapilla.

Muista kehyksistä poiketen Hapin väliohjelmia (extension points) ei suoriteta niiden rekisteröintijärjestyksessä, vaan ne ovat tapahtumasidonnaisia: saatuaan palvelinpyynnön Hapi käsittelee pyynnön elinkaaren (kuva Kuva 5) osoittamalla tavalla, jonka aikana Hapi lähettää ärsykkeitä (onRequest, onPostHandler, jne.) väliohjelmilleen. Saatuaan määritellyn ärsykkeen väliohjelma suorittaa itsensä. Käytännössä Hapi toimii siis jokseenkin samalla tavalla kuin Node.js:n oma tapahtumasilmukka (2.3.1).

extension points

1 Request object created
onRequest()
2 Find a matching route
3 Process JSONP
4 Parse cookies
onPreAuth()
5 Authenticate request
6 Read/parse payload
7 Authenticate payload
onPostAuth()
8 Validate path params
9 Validate query params
10 Validate payload
onPreHandler()
11 Run prerequisites
12 Execute handler
onPostHandler()
13 Validate response
onPreResponse()
14 Send response
'response'
15 Wait for tails
'tail'

Kuva 5. Palvelupyynnön elinkaari Hapissa. (https://twitter.com/mt_harrison/status/602731699584176129)

Hapissa reititystä voi moduloida jakamalla lisäreitit plugineihin. Lisäreittien lisäksi pluginit voivat myös lisätä palvelimeen muita toimintoja, kuten väliohjelmia. Toimiakseen plugin-moduulin tulee sisältää kaksi asiaa: register-funktion sekä register.attributes-olion. Register-funktio sisältää pluginin toiminnallisuuden, ja register.attributes pluginin lisätiedot, joista pakollisia ovat nimi sekä versionumero (koodiesimerkki Koodiesimerkki 8).

```
module.exports.register = function (server, options, next) {
  server.route({
    method: 'GET',
    path: '/',
    handler: function (request, reply) {
      reply('Welcome to my platypus page.');
    }
  });
};

module.exports.register.attributes = {
  name: 'platypus-plugin',
  version: '1.0.0'
};
```

Koodiesimerkki 8. Koodiesimerkki 5:n platypus-moduuli Hapin plugininä tehtynä.

Palvelimelle plugin ladataan server.register-metodilla (koodiesimerkki Koodiesimerkki 9). Metodille voidaan antaa lisäasetuksia. Esimerkiksi pluginin polkua voi muokata `routes`-parametrin avulla.

```
var platypus = require('./platypus')

...
server.register({ register: platypus}, {
  routes: { prefix: '/platypus' }
});
```

Koodiesimerkki 9. Koodiesimerkki 6:n mukainen moduulin lataaminen suoritettuna Hapilla.

3.5 Koa.js-kehys

Koa.js lähti käyntiin vuonna 2013 joukon Expressin kehittäjiä lähtiessä projektista, mukana mm. Expressin entinen pääkehittäjä TJ Holowaychuk. Ajatuksena Koa.js:ssä oli tehdä hyvin riisuttu kehys, joka korvaisi takaisinkutsut ECMAScript 6:ssa määritellyillä generaattoreilla [15]. Vuoden 2017 alkutalvesta Koa oli yhtä suosittu Hapin kanssa, mutta se hävisi kuitenkin Sails.js:lle ja Expressille (kuva Kuva 4).

Palvelimen luonti Koalla muistuttaa hyvin paljon tapaa, jolla se tehdään Expressissä. Suurin ero kahden välillä ovat pyynnön käsittelevät funktiot: Koassa kaikki funktiot ovat generaattoreita, ja toisin kuin Express, Koa on piilottanut palvelinpyynnön käsittelemiseen tarvittavat request- sekä response-objektit kontekstiin this-avainsanan taakse. Kontekstiin piilottamisen seurauksena väliohjelmien tarvitsema parametrilista lyhenee, ja ne tarvitsevat ainoastaan jatko-funktion kertovan next-parametrin.

Koska Koan pyynnön käsittelevät funktiot ovat generaattoreita, on pyyntöjen elinkaari jokseenkin uniikki: Jokainen pyyntö käy ensin käsittely-funktiot (väliohjelmat) läpi määrittelyjärjestyksessä kuten esimerkiksi Expressissä, mutta vastauksen saatuaan pyyntö käy väliohjelmat vielä kertaalleen läpi käänteisessä järjestyksessä suorittaen kaiken mahdollisesti jäljelle jääneet toimenpiteet.

```
var koa = require('koa');
var app = koa();

app.use(function* logger (next) {
  console.log("In comes a " + this.request.method + " to " +
this.request.url);
  this.startTime = new Date();
  yield next; // hypätään seuraavaan väliohjelmaan

  // suoritetaan virtauksen tullessa takaisin
  var timeTaken = new Date() - this.startTime;
  console.log('Time taken for request: ' + timeTaken + 'ms');
}

app.use(function* () {
  this.body = 'Hello world';
)

app.listen(3000, function() {
  console.log('Koa is listening to http://localhost:3000');
});
```

Koodiesimerkki 10. Yksinkertainen web-sovellus käyttäen Koaa.

Koan omalla termistöllä kyseessä on "virtaus". Virtaus laskee ensin alas, ja sitten nousee ylös, eli kyseessä on downstream ja upstream. Alavirrassa väliohjelmat suoritetaan yield next -käskyyn asti. Käskyn jälkeen kyseinen väliohjelma tauottuu, ja ohjelma jatkaa järjestyksessä seuraavaan funktioketjussa. Alasvirtaus jatkuu, kunnes pyyntö saa vastauksen. Yleensä tarkoitetaan, että järjestyksessä oleva funktio ei käyttänyt yield nextiä, jolloin ohjelman suuntaus kääntyy takaisinpäin ylävirraksi. Ylävirrassa funktiot käydään läpi ”alhaalta ylös”, ja suoritetaan yield nextin jälkeiset mahdolliset toimenpiteet (koodiesimerkki Koodiesimerkki 10).

```
var koa = require('koa');
var route = require('koa-route');
var app = koa();

module.exports = app;

app.use(route.get('/', function* () {
 this.body = 'Welcome to my platypus page.';
}));
```

Koodiesimerkki 11. Koodiesimerkki 5:n platypus-moduuli tehtynä Koalla ja koa-route-apukirjastolla.

Koa on todella kevyt, eikä kehys esimerkiksi tue modulaarisuutta natiivisti, vaan sille tarvitsee ladata omat apukirjastonsa. Reitittäminen saadaan lisättyä Koaan käyttämällä esimerkiksi koa-route-kirjastoa (koodiesimerkki Koodiesimerkki 11), ja irrallisia reititystiedostoja saadaan istutettua käyttäen koa-mount-kirjastoa (koodiesimerkki Koodiesimerkki 12). Syntaksisesti kirjastot ovat hyvin lähellä Expressin sisäisiä reititysmetodeja, ja oikeastaan ainut ero on, että koa-route ottaa vastaansa generaattori-funktion ja että kirjastojen nimiä joudutaan kantamaan itseään toistavasti läpi sovelluksen, toisin kuin jos funktiot löytyisivät kehyksestä itsestään.

```
var mount = require('koa-mount');
var platypus = require('/platypus');
...

app.use(mount('/platypus', platypus);
```

Koodiesimerkki 12. Koodiesimerkki 11 ladattuna koodiesimerkki Koodiesimerkki 10:n mukaiseen koodiin.

3.6 Sails.js-kehys

Sails.js on Express.js:n päälle rakennettu MVC-kehys, jonka lähtökohtana on Ruby on Railsin tapaan "convention over configuration", eli vapaasti suomennettuna käytännöllisyys ennen säädettävyyttä. Käytännössä tämä tarkoittaa, että - toisin kuin jokseenkin riisutuissa Expressissä ja Koassa - Sailsissa suurin osa toiminnallisuudesta on jo valmiina. Jopa tietokantayhteensopivuus on Sailsissa valmiina, sillä kehys sisältää Waterline-tietokantakirjaston, joka tukee suosituimpia tietokantaohjelmistoja.

Sailsissa projekti luodaan käyttämällä sisäänrakennettua projektigeneraattoria, joka ajon jälkeen luo Sailsin tarvitseman kansiorakenteen ja yksinkertaisen index-sivun. Hyötynä tästä on, että ohjelmoija pääsee nopeasti hommiin. Haittapuolena Sailsin rakenne on

hyvin jäykkä, eikä sen muokkaaminen omia mieltymyksiä vastaavaksi ole helppoa, tai edes suositeltavaa.

```
// tiedosto /api/controllers/HelloController.js

module.exports = {
 // automaattiseksi poluksi muodostuu /hello/world
   world: function (req, res) {
      return res.send('Hello world!');
   }
}
```

Koodiesimerkki 13. Yksinkertainen Sails.js controller-moduuli

Sails luo reitit automaattisesti sen mukaan, millaisia Controller-objekteja on /api/controllers/-kansiossa, ja niiden sisällön mukaisesti reittejä (koodiesimerkki Koodiesimerkki 13). Automaattisesti muodostetut reitit ovat muotoa "/objektin nimen alkuosa/objektin ominaisuuden nimi", ja vastaavat oletuksena kaikkiin HTTP-verbeihin. Automaattista reitinmuodostusta voi muokata haluamakseen muokkaamalla Sailsin sisältämän Blueprint API:n asetuksia tiedostossa /config/blueprint.js.

Väliohjelmia Sailsissa on kahdenlaisia: Sailsille itselleen tehtyjä, joita Sails kutsuu "säännöiksi" (policy), sekä Expressille tehtyjä väliohjelmia. Sailsin sääntöjä tehdessä on tärkeää määritellä kukin erilliseksi moduuliksi /api/policies-kansioon. Säännöt eivät tule automaattisesti käyttöön, vaan ne pitää erikseen asettaa controller-kohtaisesti /config/policies.js-tiedostossa. Expressille tehdyt väliohjelmat saa sellaisenaan käyttöön lisäämällä ne joko /config/http.js-tiedostoon tai importoimalla säännöiksi.

3.7 Päätelmät

Sinatramaisten kehysten (eli vertailusta kaikki paitsi Sails) antamien vapauksien välillä oli huomattavia eroja – Koa.js antoi eniten vapautta olemalla mahdollisimman riisuttu, ja antamalla ohjelmoijalle täydet mahdollisuudet mikromanageroida ominaisuuksia halujensa mukaan. Hapi antoi vapautta, mutta kuitenkin sisälsi kolmikosta eniten sääntöjä ja opittavaa. Express oli väliinputoaja, joka sisälsi suuren osan tarpeellisimmista ominaisuuksista, mutta oli kuitenkin karsittu.

Railsmaisista kehyksistä ainoana oli mukana Sails.js. Sails vaatii opeteltavaa mutta on tästä huolimatta hurmaavan aloittelijaystävällinen. Sivujen tekeminen on tehty mahdollisimman helpoksi, ja soveltuu erinomaisesti, jos tahtoo kastella varpaitaan Node.js:ssä ennen kuin sukeltaa ominaisuuksiltaan kevyempään kehykseen kuten Expressiin.

Sinatramaiset kehykset ovat selvästi se, mihin kannattaisi pyrkiä, jos haluaisi saada mahdollisimman paljon irti Node.js:n keveydestä – kaikki paitsi Hapi. Vaikka Hapi antaa vapauksia, on sen opettelu vähintään yhtä raskasta kuin railsmaisen Sailsin. Hapi ei kuitenkaan ole rakenteeltaan yhtä selkeä kuin Sails. Sinatramaisista Express tuntuu todella hyvältä vaihtoehdolta – paremmalta kuin Koa. Koa sisältää kenties jopa liikaa mikromanagerointia, ja vaikka generaattorit ovatkin periaatteessa siistejä, niin käytännössä ne tuntuvat turhalta kikkailulta, jotka uutuudenviehätyksen karistessa lähinnä mutkistavat asioita. Express on hyvä kompromissi keveyden ja käytettävyyden välillä, ja on selkeää, miksi se on suosituin. Sails on myös hyvä vaihtoehto, jos kehyksen jäykkyys ei haittaa.

4 Sivupohjat

Kokonaisen web-sivuston rakentamiseen tarvitaan kehyksen lisäksi myös näytettävä sisältö. Yksinkertaisimmillaan sisältö voi olla pelkkää tekstiä, mutta yleensä tekstiä on muotoiltu erilaisilla HTML-tägeillä, ja koristettu CSS:llä. Dynaamisuutta sivuille voi lisätä JavaScriptillä. Kuitenkin sivuston paisuessa suureksi sen ylläpito hankaloituu, ja tarvitaan sivupohjien tarjoamaa abstrahointia.

Sivupohjista puhuessa yleensä tarkoitetaan muotteja, joiden avulla luodaan näytettävä web-sivu, eli MVC-mallin View. Yksinkertaisimmillaan sivupohjat ovat staattisia ja pääosin HTML:llä kirjoitettuja, mutta joihin lisätään kontekstiriippuvaista sisältöä, esimerkiksi reitityksen mukaan vaihtuva otsikko. Usein paikat, joihin kontekstiriippuvainen sisältö asetetaan, merkitään sivupohjalle tunnusomaisilla erikoismerkeillä.

Kenties hyödyllisin sivupohjien ominaisuus on kuitenkin niiden mahdollistama koodin uudelleen käyttö: abstrahoimalla samaa sivupohjatiedostoa voi käyttää hyvinkin erilaisissa asiayhteyksissä. Ohjelmoija voi esimerkiksi määritellä alkupohjan, joka sisältää asioita, jotka löytyvät jokaiselta alasivulta, kuten yleisimmät riippuvuudet, ja navigaatiopalkin. Lapsipohjat sitten perivät alkupohjan, ja yliajavat valitut kentät. Koodin uudelleenkäyttöä

voi useimmiten myös harrastaa pilkkomalla koodit pieniin pätkiin (snippet), jotka sitten injektoidaan muun pohjan joukkoon tarvittaessa.

Sivupohjat voi yleensä jakaa kahdenlaisiin – palvelimessa ja selaimessa toimiviin. Suurin ero pohjatyyppien välille tulee siinä, kummassa päässä pyyntöä generoimisen taakka on, mutta eroavat myös siinä, paljonko palvelimen tarvitsee lähettää dataa selaimelle.

Toimintaympäristön lisäksi sivupohjat voidaan jakaa sisältämiensä ominaisuuksien mukaan logiikallisiin ja logiikattomiin sivupohjiin: logiikattomista kenties tunnetuin esimerkki on Mustache. Logiikattomuus tarkoittaa, ettei sivupohja itsessään sisällä koodia tai tee valintoja. Tästä seuraa, että työnjako eri komponenttien välillä on jaettu selkeästi (modulaarisuus), ja että sivupohjaa itsessäänkin on selkeämpi lukea. Seuraavassa osiossa on vertailua eräiden suosittujen sivupohjien välillä.

4.1 Pug-sivupohja

Pug on minimalistinen sivupohja, joka käyttää kokonaan omaa kieltään abstrahoidakseen HTML-tagit pois jättäen pelkästään tagien avainsanat. Sulkutagejä ei tule ollenkaan, ja sisällön ryhmittäminen tapahtuu Python-ohjelmointikielen tapaan sisennyksien avulla.

Pug sisältää paljon ominaisuuksia, kuten JavaScript-osioiden upottamisen sivupohjaan (koodiesimerkki Koodiesimerkki 14) sekä perinnöllisyyden. Pugissa modulaarisuus toimii enimmäkseen import-lausekkeen kautta, jolla Pugin pätkiä voi lisätä isäntänä toimivaan sivupohjaan. Funktioita Pug kutsuu mix-ineiksi. Mix-init määritellään sivupohjaan ja sitten kutsutaan kuten funktioita, tarvittaessa annetaan parametreja. Pug on aiemmin tunnettu nimellä Jade.

```
//pala pug-koodia
html
  //määritetään listaelementtien määrä JavaScriptillä
  - var x = 5;
  div
    ul
    //tehdään tarvittava määrä elementtejä JavaScriptin for-silmukalla
      - for (var i=1; i<=x; i++) {
        li Hello World!
      - }

/*
```

```
 ulossyöte:
<html>
  <div>
    <ul>
      <li>Hello World!</li>
      <li>Hello World!</li>
      <li>Hello World!</li>
      <li>Hello World!</li>
      <li>Hello World!</li>
    </ul>
  </div>
</html>
*/
```

Koodiesimerkki 14. Pug-sivupohjamoottorilla tehty sivu.

4.2 Handlebars-sivupohja

Handlebars perustuu suosittuun logiikattomaan Mustache-sivupohjamoottoriin. Etuna Handlebarsissa on mm. että sivupohjat voi generoida etukäteen, ja lisätä sivuun JavaScriptinä, mikä nopeuttaa sivujen muodostusta. Verrattuna Mustacheen sivut generoituvat tuplasti nopeammin.

```
//rekisteröidään apufunktio ja määritellään konteksti-objekti
var context = {
  items: [
    {name: "Handlebars", emotion: "love"},
    {name: "Mustache", emotion: "enjoy"},
    {name: "Ember", emotion: "want to learn"}
  ]
};

Handlebars.registerHelper('agree_button', function() {
  var emotion = Handlebars.escapeExpression(this.emotion),
      name = Handlebars.escapeExpression(this.name);

  return new Handlebars.SafeString(
    "<button>I agree. I " + emotion + " " + name + "</button>"
  );
});
```

Koodiesimerkki 15. Apufunktion rekisteröiminen Handlebarsissa.

Toisin kuin Pug, Handlebars ei tue JavaScript-koodin upottamista sivupohjaan. Sen sijaan monimutkaisempi toiminnallisuus suoritetaan apufunktioiden avulla, jotka rekisteröidään käyttäen Handlebarsin `registerHelper`-funktiota ennen sivupohjan renderöintiä (koodiesimerkki Koodiesimerkki 15). Rekisteröimisen jälkeen apufunktiota voi kutsua sivupohjassa (koodiesimerkki Koodiesimerkki 16).

```
// Handlebars-koodi joka käy kontekstiin määritellyn listan läpi ja
// kutsuu jokaisen esineen kohdalla apufunktiota
<ul>
  {{#each items}}
  <li>{{agree_button}}</li>
  {{/each}}
</ul>

// generoitu HTML-koodi
<ul>
  <li><button>I agree. I love Handlebars</button></li>
  <li><button>I agree. I enjoy Mustache</button></li>
  <li><button>I agree. I want to learn Ember</button></li>
</ul>
```

Koodiesimerkki 16. Handlebars-sivupohjamoottorilla tehty sivu.

Handlebars tukee modulaarista sivupohjien muodostusta osituksien muodossa (partial). Käytännössä nämä toimivat kuten import-lauseet Pugissa, eli pääpohjaan lisätään pienempiä koodilohkoja. On kuitenkin huomioitava, että toisin kuin pääpohja, etukäteen generoidessa osituksia ei tehdä valmiiksi vaan Handlebars rekisteröi ne apufunktioina, jotka sitten suoritetaan ajon aikana.

4.3 React-sivupohja

React on sivupohjamoottori, joka pyörii selaimessa. Reactin pääasiallinen tarkoitus on tehdä esim. AJAX-kutsuista johtuvasta käyttöliittymän päivittämisestä mahdollisimman nopeaa. Tämä tapahtuu siten, että React päivittää vain ne osat, joissa on ollut muutoksia: esimerkiksi jos käyttöliittymässä näkyvän listan alkioista vain yksi on muuttunut, niin kokonaista listaa ei renderöidä uudestaan vaan vain muuttunut alkio.

Tyypillinen React-sivupohja on rakennettu pääosin HTML:stä, jossa dynaamiset osat ovat JavaScriptillä tai JSX:llä rakennettuja React-luokkia (koodiesimerkki Koodiesimerkki 17). JSX on XML:n kaltainen kieli, jonka suurin ero XML:n nähden on, että toisin kuin XML, voi se sisältää raakaa HTML-koodia. Koska React on pääosin HTML:ää ja toimii selaimessa eikä palvelimella, niin sitä voi helposti käyttää palvelinpuolen sivupohjamoottorien, kuten Pugin, kanssa saumattomasti.

```
//tarvittavat riippuvuudet
<script src="https://unpkg.com/react@15.3.1/dist/react.js"></script>
<script src="https://unpkg.com/react-dom@15.3.1/dist/react-
dom.js"></script>
<script src="https://unpkg.com/babel-
core@5.8.38/browser.min.js"></script>

<div id="container">
<!-- Kohde -->
</div>

<script type="text/babel">
//aloitetaan muodostamalla React-luokka
var Hello = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});
//renderöidään luokka
ReactDOM.render(
  <Hello name="World" />, //huom. muoto (JSX)
  document.getElementById('container')
);
</script>

//tulos
<div id="container">
  <div data-reactroot="">
  <!-- react-text: 2 -->Hello <!-- /react-text --><!-- react-text: 3 -
->World<!-- /react-text -->
  </div>
</div>
```

Koodiesimerkki 17. Reactillä tehty Hello World! -esimerkki.

5 Toteutus

5.1 Kansiorakenne

Ensimmäinen askel projektin luontiin on kansiorakenne. Yleisesti hyväksytty käytäntö on pitää projektin juurikansiossa app.js-tiedostoa, jonka ajamalla node.js-palvelin poljetaan käyntiin. Käytettävästä kehyksestä riippuen käynnistystiedosto (app.js) voi myös sisältää alkumäärittelyjä, kuten tietokantapalvelinyhteyden muodostamisen ja väliohjelmien (middle-ware) käyttöönoton.

Juurikansio sisältää yleensä käynnistystiedoston lisäksi kehityksen apuohjelmien projektin kattavia asetustiedostoja, kuten esim. paketinhallintatyökalu npm:n tarvitsema package.json-tiedoston ja laadunvalvontatyökalu JSHintin .jshintrc-asetustiedoston.

Varsinainen palvelinlogiikka sijaitsee yleensä alakansioissa. Joitain yleisiä ovat seuraavat:

- Config: sisältää yleensä kootusti käynnistyksessä tarvittavat asetukset, esimerkiksi tietokannan tiedot on hyvä laittaa tähän kansioon. Usein asetustiedostojen muoto on JSON.
- Libs: sisältää npm:n ulkopuoliset apukirjastot. Usein käyttäjän itsensä tekemiä.
- Models: sisältää tietokannan oliomäärittelyt. Oliomäärittelyihin on useimmissa JavaScriptin tietokantakirjastoissa mahdollisuus sisällyttää usein käytettyjä apufunktioita. Esimerkiksi jos sovellus usein tarvitsee olioiden määrän, on määrän selvittävä funktio hyvä sisällyttää olion määrittelyyn.
- node_modules: liitännäistenhallintasovellus npm:n kautta yleensä ylläpidettävä kansio. Sisältää sovellukselle tarpeellisten liitännäisten lähdekoodin.
- Public: sisältää julkiset tiedostot, jotka näkyvät palvelimen ulkopuolelle. Useimmiten ovat JavaScript-, CSS- ja kuvatiedostoja.
- Routes: sisältää kehyksen reititysten määrittelyn.
- Test: sisältää mahdolliset yksikkötestit (unit test).

5.2 Riippuvuudet

Ohjelmistoprojekteissa on yleensä ulkoisia riippuvuuksia, joiden hallitseminen voi olla hankalaa. Node.js:ssä asia on ratkaistu paketinhallintatyökalu npm:llä, joka asennetaan Node.js:n mukana. Projektin kirjastoriippuvuudet merkitään Node.js-projektin package.json-asetustiedostoon, jonka jälkeen npm osaa ladata riippuvuudet pilvestä asennuskäskyllä `npm install`. Jos projektissa ei vielä ole asetustiedostoa, voi npm:ä pyytää avustamaan sen tekemisessä `npm init` -komennolla. Npm:llä voi asetustiedostossa jo mainittujen kirjastojen lisäksi asentaa siellä mainitsemattomia kirjastoja: tämä tapahtuu antamalla asennuskäskyn lopuksi asennettavan kirjaston nimen. Npm:n voi laittaa kirjaamaan asennettavan, asetustiedostosta löytymättömän, kirjaston asetustiedostoon lisäämällä asennuskäskyyn vielä `--save`-parametrin.

Seuraavassa on joitain yleisimmin käytettyjä package.json-asetustiedoston attribuutteja:

- Name: projektin nimi (pakollinen).
- Version: projektin nykyinen versio (pakollinen).
- Dependencies: projektin kirjastoriippuvuudet.

Jos käyttäjällä on omia kirjastoja, joista projekti on riippuvainen, voi käyttäjä julkistaa ne npm:n, ja siten asentaa muiden riippuvuuksien mukana. Oletuksena kaikki npm:ssä julkistetut kirjastot ovat julkisia, mutta npm on myös tarjonnut mahdollisuutta henkilökohtaisille kirjastoille lisämaksua vastaan.

5.3 Reititys

Reitityksessä on kyse index-sivusta haarautuvien alasivujen polkujen määrittelystä. Usein tähän kuuluu myös polulle ominaisten toimintojen määrittely: esim. admin-sivuille johtava polku voi sisältää käyttäjän johdattamisen ensin kirjautumissivulle, jonka jälkeen hän vasta pääsee alkuperäisen määränpäähänsä.

Reititystiedosto(t) sijaitsevat hyvin usein omassa kansiossaan routes. Kuitenkin jos projekti on erittäin pieni, voivat ne hyvin sijaita myös tyypillisesti projektin juurikansiossa olevassa palvelimen käynnistystiedostossa. Jos reititystiedostot ovat omassa kansiossaan, on hyvin yleinen suositus laittaa jokainen eri polku omaan tiedostoonsa polkurakenteen selkiyttämiseksi.

Poikkeuksena yleisestä käytännössä railsmaisissa kehyksissä (ks. luku 3.2) kansiorakenne kuitenkin on jokseenkin uniikki: esimerkiksi Sails.js:ssä (ks. luku 3.6) reititystiedostot ovat ns. kontrollereja, jotka sijaitsevat /api/controllers-kansiossa. Kehyksen mukanaan tuoma automaatio myös kannustaa tekemään isoja reititystiedostoja, jotka sisältävät useita polkuja.

Reititystiedoja, ja niiden sisältämiä polkuja, on yleisesti kannustettu olemaan toiminnoiltaan mahdollisimman yksinkertaisia. Monimutkaisten välitoimintojen oikeata olinpaikkoja ovatkin väliohjelmat, sillä muusta palvelinkoodista eristettyinä ne ovat yleensä helpompia testata ja ylläpitää.

5.4 Sivupohjat

Sivupohjavaihtoehtoja (ks. luku 5) on kaikille Node.js-kehyksille useampia, ja käytännössä useimmat sivupohjat ovat yhteensopivia kaikkien sivupohjia tukevien kehyksien kanssa. Kuitenkaan kaikki kehykset eivät tue sivupohjia natiivisti, vaan käyttöönottoa varten tarvitse asentaa erillinen liitännäinen. Näin on esimerkiksi Hapin (ks. luku 4.4) ja Koa.js:n (ks. luku 4.5) kohdalla.

Yleensä käyttöönotto noudattaa tiettyä kaavaa: Ensin asennetaan sivupohjamoottori, jonka jälkeen se importoidaan kehyksen, tai sivupohjia hallitsevan liitännäisen, `engine`-asetukseen. Importoinnin lisäksi on tärkeää kertoa kehykselle/liitännäiselle kansio, missä raa'at sivupohjatiedostot ovat. Lopuksi kutsutaan sivupohjan render-metodia sivun luomiseksi.

Poikkeuksiakin sivupohjien käyttöönottoon on: esimerkiksi Sails.js käyttää sisäisesti consolidate.js-moduulia. Moduulin käyttö näkyy kirjoitettavan koodin määrässä: ohjelmoijan ei tarvitse kuin asentaa sivupohjamoottori, ja sitten asetustiedostossa `/config/views.js` kertoa käyttävänsä sen nimistä moottoria, ja kenties asettaa kansio, jossa raa'at sivupohjat ovat, jos oma käyttämä poikkeaa oletuskansiosta.

5.5 Omien kirjastojen luonti

Yleisenä filosofiana Node.js:ssä on modulaarisuus, eli pyritään tekemään jokaiselle toiminnallisuudelle oma moduulinsa. Uudet moduulit on hyvä tallentaa omaan kansioonsa projektin juuresta nähden, esim. libs on kansiolle hyvä nimi. Jokaisen uuden moduulin perustana on *module.exports*, sillä vain siihen lisätyt rakenteet näkyvät moduulitiedoston ulkopuolelle.

Yleisesti suositeltua on tiedostoissa laittaa tärkeimmät asiat ylimmäksi: Node.js:ssä ne ovat moduulin riippuvuudet sekä muuttujamääritelmät. Moduulin ulkopuolelle näkyvien rakenteiden, eli exporttien, esittelyn sijoittaminen mahdollisimman ylös tekee moduulin tarjoamien asioiden näkemisestä helppoa, ja niiden olisikin hyvä olla seuraavina. Jos export on funktio, tulee sen määritelmä joko laittaa juuri ennen export-määritelmää, tai etenkin, jos funktio on rivimäärältään suuri, export-määritelmien jälkeen. Viimeisenä tulee muu moduuliin mahdollisesti liittyvä koodi.

Riippuvuudet määritellään Node.js:ssä require-lauseilla, esimerkiksi jos haluaa tuoda moduuliin `node_modules`-kansiossa olevan path-kirjaston, ja asettaa sen muuttujaan path, tulee kirjoittaa `var path = require('path')`. Huomattavaa on, että jos riippuvuus sijaitsee muualla kuin Node.js:n oletuksena käyttämässä moduulikansiossa, tulee riippuvuuden sijainti kertoa suhteessa moduuliin. Esimerkiksi: jos on libs-kansiossa itsetehty kuvitteellinen moduuli, jonka nimi on platypus.js, ja haluaa juurikansiossa olevan app.js-tiedoston käyttävän sitä, tulee moduuli tuoda komennolla `var platypus = require('./libs/platypus')`.

5.6 Testaus ja laadunvarmistus

5.6.1 Yksikkötestaus

Suosituimmat Node.js:n kanssa yhteensopivat testikirjastot kirjoittamishetkellä ovat Mocha ja Jasmine. Kenties suurin ero kahden välillä on, että toisin kuin Jasmine, Mocha itsessään tarjoaa vain testirakenteen, eli assert- sekä mocking-toiminnallisuutta varten täytyy ottaa käyttöön ulkopuolisia kirjastoja. Yleisesti ottaen Jasmine sisältää kaikki työkalut, mutta Mocha antaa ohjelmoijalle vapaammat kädet, miten haluaa asiat tehdä.

Joitain suosittuja assert-kirjastoja ovat should.js sekä Chai. Omassa projektissani käytin Chaita, sillä se on monipuolinen ja myös useissa esimerkeissä käytetty. Käytännössä suurin osa assert-kirjastoista kuitenkin toimii jokseenkin samalla tavalla, ja suurimmat erot kirjastojen välillä ovat niiden sisältämän syntaksisen sokerin määrässä. Myös mocking-kirjastoja on useampia, joita käyttää Mochan kanssa. Niistä tunnetuin ja yleensä esimerkeissä käytetty on Sinon.

```
describe('User', function() {
  describe('#save()', function() {
    it('should save without error', function(done) {
      var user = new User('Luna');
      user.save(done);
    });
  });
});
```

Koodiesimerkki 18. Yksinkertainen yksikkötesti rakennettuna käyttäen BDD-termejä.

Kummatkin kirjastot, Mocha ja Jasmine, tukevat done-funktionsa avulla Node.js-tyylin mukaista virheiden tarkistusta (koodiesimerkki Koodiesimerkki 18). Esimerkkikoodissa user.save on funktio, joka kutsuu valmistuessaan paluu-funktiota (done). Kutsuttu funktio

tulkitsee saamistaan parametreista, onnistuiko toimenpide, ja määrittää testin lopputuloksen. Kummassakin kirjastossa on aikaraja, jonka aikana yksikkötestin tulee kutsua paluu-funktiota. Muutoin kirjasto katsoo testin epäonnistuneen.

Testit rakentuvat TDD-termejä käyttäen niin, että ylimpänä on testijoukko, jonka sisällä voi olla suiteja ja/tai testejä. BDD-tyylillä avainsana ”suite” on ”describe”, ja ”test” on ”it” (koodiesimerkki Koodiesimerkki 18). Kirjastoista vain Mocha tukee TDD-tyyliä, mutta kummatkin tukevat BDD-tyyliä. Kummatkin kirjastot tukevat tavallisimpia koukkuja (hooks), joilla voidaan esimerkiksi määritellä alkutoimet ennen testiä.

5.6.2 Koodin tyylin tarkistus

Tyylintarkistuksessa tavoitteena on löytää testattavasta koodista epäkäytännölliset, tai muutoin parhaita käytäntöjä vastaan sotivat koodin pätkät. JavaScriptille siihen on olemassa joukko kirjastoja, joista yksi suosituimmista on JSHint. JSHintin asennus käy npm:n kautta komennolla `npm install jshint`. Oletusarvoisesti JSHint pitää ajaa manuaalisesti komentoriviltä, mutta sen saa myös ajamisen voi myös integroida osaksi Grunt-rutiinia. Lisäksi JSHintin voi asentaa joihinkin tekstieditoreihin plug-ininä.

Asioita joihin JSHint kiinnittää huomiota voi räätälöidä tekemällä .jshintrc-asetustiedostoja. Esimerkkinä asetustiedosto, joka varoittaa käyttämättömistä muuttujista sekä muuttujista, joiden sisennysmuuttuja ei ole kolme välimerkkiä pitkä, eikä anna varoitusta Mochan tai Node.js:n avainsanoista:

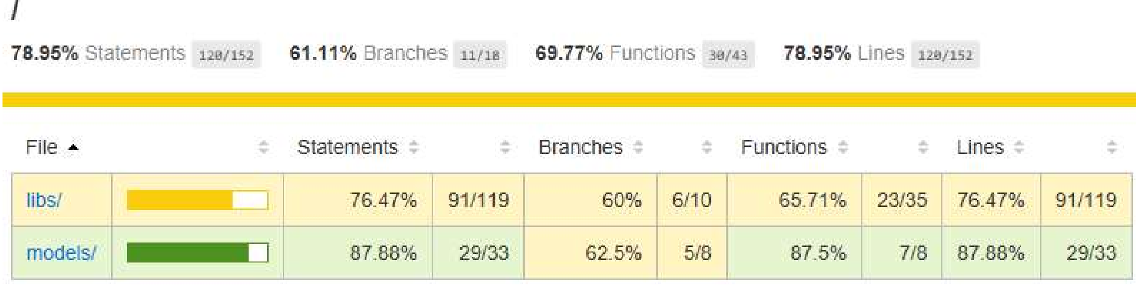
```
{
   "indent": 3,
   "unused": "vars",

   "mocha": true,
   "node": true
}
```

Ohjelmoijalla on varsin vapaat kädet sen suhteen, mihin sijoittaa JSHintin asetusmuutoksensa, ja hän voi esimerkiksi tehdä pääkansioon yhden pääasetustiedoston, josta löytyy projektin kattavat asetukset, ja sitten lapsikansioihin omat tiedostot, joiden asetukset yliajavat pääasetustiedostoa erityistarpeita lapsikansion sisältöä mukaillen. Asetustiedostojen tekemisen lisäksi käyttäjä voi myös hienosäätää asetuksia itse kooditiedostojen sisällä.

5.6.3 Kattavuuden testaus Istanbulilla

Yksikkötestien koodikannan kattavuudesta voi olla hankala pitää kirjaa. Hyvä aputyökalu tähän on kattavuudentestain. Node.js:lle eräs suosituimmista kirjastoista kattavuuden testaukseen on Istanbul, jota käytän tässä esimerkkinä.

Istanbulin käyttö on yksinkertaisimmallaan helppoa. Esimerkiksi jos käyttää Windowsia, ja yksikkötestaustyökalu on Mocha, niin Npm:llä asennuksen jälkeen raportin saamiseen riittää komento `istanbul cover node_modules/mocha/bin/_mocha -- -R spec`, jossa kaksoisviivaa edeltävä polku on Mochan ajotiedoston sijainti, ja jälkeinen osa Mochan parametreja. Jasmine 2:lla raportin luominen tapahtuu jotakuinkin samalla tavalla. Istanbulin luoma raportti löytyy ajokansion alakansiosta `coverage`.



/

78.95% Statements 120/152 61.11% Branches 11/18 69.77% Functions 30/43 78.95% Lines 120/152

File		Statements		Branches		Functions		Lines	
libs/		76.47%	91/119	60%	6/10	65.71%	23/35	76.47%	91/119
models/		87.88%	29/33	62.5%	5/8	87.5%	7/8	87.88%	29/33

Kuva 6. Esimerkki Istanbulin luomasta raportista. Kuvassa yleiskatsaus testattujen kansioiden testien kattavuudesta.

Istanbulin raportit ovat hyvin helppolukuisia ja yksinkertaisia (kuva Kuva 6). Selitän seuraavaksi joitain kansiokohtaisista raporteista löytyviä sarakkeita. Huomattavaa on, että jokainen sarake sisältää ensin prosentuaalisen määrän, ja sitten lukumääräisen:

- Statements – yksikkötestien kattamat lauseet.
- Branches – yksikkötestien kattamat ehtolauseet.
- Functions – yksikkötestien kattamat funktiot.
- Lines – yksikkötestien kattamat rivit.

all files / libs/ PipedRequest.js

84.62% Statements 11/13 100% Branches 0/0 75% Functions 3/4 84.62% Lines 11/13

```
1       'use strict';
2
3   1×  var request = require('request');
4
5   1×  module.exports = function PipedRequest(uri, pipeTarget) {
6
7   2×     var self = this;
8   2×     this.targetUri = uri;
9   2×     this.pipeTarget = pipeTarget;
10
11  2×     this.pipe = function() {
12  2×        var stream = request.get(self.targetUri);
13  2×        stream.on('error', handleError);
14  2×        stream.on('response', function onResponse(res) {
15  2×           res.pipe(self.pipeTarget);
16           });
17        };
18
19  1×     function handleError(err) {
20           console.log(err, err.stack);
21           return process.exit(1);
22        }
23     };
```

Kuva 7. Esimerkki tiedostokohtaisesta raportista Istanbulissa.

Tiedostokohtaiset raportit (kuva Kuva 7) tarjoavat kansiokohtaisia raportteja yksityiskohtaisempaa tietoa. Niistä löytyy esimerkiksi tieto, montako kertaa testien aikana kukin rivi on luettu (kuva Kuva 7 kertamäärä rivinumeron viereisessä sarakkeessa), ja mitä alafunktioita ei ole suoritettu testien aikana ollenkaan (kuva Kuva 7 punaisella merkityt rivit).

5.6.4 Grunt-automatisaatio

Kaikkien testaustyökalujen ajaminen yksi kerrallaan voi olla pidemmän päälle hyvin työlästä. Ohjelmoija voi kuitenkin käyttää apunaan automaatiotyökalua, kuten Gruntia. Gruntin ajaminen tapahtuu komentoriviltä, ja sen nimi npm:ssä onkin grunt-cli, jossa CLI tarkoittaa komentorivirajapintaa.

Kenties hieman erikoisesti Grunt tarvitsee kuhunkin ajettavaan työkaluun oman lisäosansa, jotka voi ladata Gruntin tapaan npm:n avulla. Lista olemassa olevista lisäosista on helppo löytää esimerkiksi Gruntin kotisivuilta.

Grunt-toimintojen määrittely tapahtuu tiedostossa Gruntfile.js, eli ns. Grunt-tiedostossa, joka tavallisesti sijaitsee projektin juurihakemistossa. Grunt-tiedosto on Node.js-moduuli, josta näky ulos funktio, jonka ainoa parametri on instanssi Gruntista. Moduulin sisältämä funktio sisältää yleensä seuraavat askeleet (koodiesimerkki Koodiesimerkki 19):

1) tarvittavien lisäosien lataaminen
2) lisäosien konfigurointi
3) Grunt-tehtävien määrittely, ja lisäosien sitominen niihin.

```
module.exports = function configGrunt(grunt) {

   // ladataan lisäosat Mochaa ja JSHintiä varten
   [
      'grunt-mocha-test',
      'grunt-contrib-jshint',
   ].forEach(function (task) {
      grunt.loadNpmTasks(task);
   });

   //lisäosien konfiguraatio
   grunt.initConfig({

      mochaTest: {
         unit: {
            options: {
               reporter: 'spec',
               ui: 'tdd'
            },
            src: 'test/tests-*.js'
         }
      },

      jshint: {
         options: {
            jshintrc: true,
            force: true
         },
         app: ['app.js', 'public/js/**/*.js', 'libs/**/*.js'],
         qa: ['Gruntfile.js', 'test/**/*.js']
      },
   });

   // määritellään komento “test”, joka ajaa Mochan ja JSHintin
   grunt.registerTask('test', ['mochaTest', 'jshint']);
};
```

Koodiesimerkki 19. Esimerkkitoteutus Grunt-tiedostosta.

6 Yhteenveto

Insinöörityön tavoitteena oli tutkia Node.js:ää ja JavaScriptiä yleisesti, ja vertailla erilaisia vaihtoehtoja sen osalta, miten sovellus voitaisiin Node.js:llä rakentaa. Tehtävä oli jokseenkin haastava, sillä tieto tuntui olevan vuonna 2015 hyvin hajautunutta, ja vaikka kirjoja oli jonkin verran, niin harva tuntui menevän pintaa syvemmälle. Huomattava tiedonlähde olikin yksityisihmisten blogaukset aiheesta.

Insinöörityön kirjoittamisen aloitin aika kauan sen jälkeen, kun olin jo aloittanut ohjelmointiprojektin tekemisen. Tuntui hieman hassulta vertailla kehyksiä, kun olin jo valinnut Expressin alustakseni. Toisaalta se, että olin jo tehnyt jonkin verran työtä Node.js:n parissa antoi myös apuja vertailun kannalta tärkeiden asioiden löytämiseen. Vertailun lopuksi oli tyydyttävää, että ensisijainen valintani osoittautui vertailussa myös parhaaksi.

Sivupohjista en aluksi tiennyt nimeltä yhtäkään, ja vertasin niitä keskenään jo ennen kuin aloitin projektini. Aluksi kaavailin ottavani osioon nykyistä enemmän sivupohjia, mutta päädyin ottamaan projektissa käyttämäni Pugin, ja muutaman sellaisen, joiden tunsin olevan relevantteja. Projektini piti sisällään paljon staattisia sivuja, ja React tuntui sen takia väärältä työkalulta. Pugissa viehätti eniten sen virtaviivaisuus, mutta toisaalta myös jousto, jota se tarjosi. Handlerbars tukee useampaa palvelinratkaisua, ja olisi luultavasti ollut hyvä valinta, jos en olisi ollut alustasta varma.

Alun perin Toteutus-osiossa oli tarkoitus puhua projektistani, ja miten tein jokaisen osion. Kuitenkin tuntui paljon luonnollisemmalta käydä toteutuksessa tarvittavat osat läpi yleisellä tasolla.

Merkittävin asia jonka sivuutin kokonaan, oli tietokantakerros. Käytin insinöörityötä varten tekemässäni ohjelmointiprojektissa Sequelize-kirjastoa ottaakseni yhteyttä tietokantaan, mutta tuntui epäreilulta puhua vain tästä ratkaisusta. Käytin projektissani tietokantayhteyttä niin vähän, ettei se välttämättä ole tarpeeksi relevantti web-sivun rakentamisessa tarvittavan tutkimustyön määrään nähden.

Lähteet

1 McCarthy, Kevin. 2011. Node.js Interview: 4 Questions with Creator Ryan Dahl. Verkkodokumentti. <http://bostinno.streetwise.co/2011/01/31/node-js-interview-4-questions-with-creator-ryan-dahl/>. 31.1.2011. Luettu 11.5.2015.

2 Finley, Klint. 2014. Future of Popular Coding Tool in Doubt After It Splits in Two. Verkkodokumentti. <http://www.wired.com/2014/12/io-js/>. 12.3.2014. Luettu 14.5.2015.

3 Brown, Ethan. 2014. Web Development with Node and Express. Sebastopol: O'Reilly.

4 Ogier, Alex, 2016. Node performance 2016: Hapi, Express.js, Restify and Koa. Verkkodokumentti. <https://raygun.com/blog/2016/06/node-performance/>. 2.7.2016. Luettu 6.4.2017.

5 Pressler, Ron. 2014. Little's Law, Scalability and Fault Tolerance: The OS is your bottleneck (and what you can do about it). Verkkodokumentti. <http://blog.paral-leluniverse.co/2014/02/04/littles-law/>. 4.2.2014. Luettu 11.5.2015.

6 Takada, Mikito. 2011. Understanding the node.js event loop. Verkkodokumentti. <http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/>. 1.2.2011. Luettu 14.5.2015.

7 Jiang, Eric. 2014. The emperor's new clothes were built with Node.js. Verk-kodokumentti. <http://notes.ericjiang.com/posts/751>. 4.6.2014. Luettu 13.5.2015.

8 Swenson-Healey, Erin. 2013. The JavaScript Event Loop: Explained. Verkko-dokumentti. <http://blog.carbonfive.com/2013/10/27/the-javascript-event-loop-ex-plained/>. 27.10.2013. Luettu 6.4.2017.

9 Inheritance and the prototype chain. 2015. Verkkodokumentti. <https://develo-per.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_proto-type_chain>. 8.1.2017. Luettu 19.1.2017.

10 Wasson, Mike. 2013. ASP.NET - Single-Page Applications: Build Modern, Re-sponsive Web Apps with ASP.NET. Verkkodokumentti. <https://msdn.micro-soft.com/en-us/magazine/dn463786.aspx>. 4.11.2013. Luettu 17.1.2017.

11 Google Trends. 2017. Verkkodokumentti. <https://trends.google.fi/trends/ex-plore?date=2010-01-01%202017-02-06&q=hapi%20node,node%20ex-press,sails.js,node%20koa>. 6.2.2017. Luettu 6.2.2017

12 Google Trends. 2017. Verkkodokumentti. <https://www.google.fi/trends/ex-plore?date=2010-01-01%202017-02-06&q=hapi%20node,sails.js,node%20koa>. 6.2.2017. Luettu 6.2.2017.

13 Tsang, Al. 2014. TJ Holowaychuk Passes Sponsorship of Express to StrongLoop. Verkkodokumentti. <https://strongloop.com/strongblog/tj-holowaychuk-sponsorship-of-express/>. 29.7.2014. Luettu 6.4.2017.

14 Hammer, Eran. 2012. hapi, a Prologue. Verkkodokumentti. <https://hueniverse.com/hapi-a-prologue-842e166fdd08>. 20.12.2012. Luettu 26.11.2015.

15 Sanderson, Steve. 2013. Experiments with Koa and JavaScript Generators. Verkkodokumentti. <http://blog.stevensanderson.com/2013/12/21/experiments-with-koa-and-javascript-generators/>. 21.12.2013. Luettu 6.4.2017.