REST-rajapinnan toteutus Flask-sovelluskehyksellä

Petrus Hurtig

Haaga-Helia ammattikorkeakoulu

Tietojenkäsittelyn tutkimusohjelma

AMK-opinnäytetyö

2023

Tiivistelmä

Tekijä(t)
Petrus Hurtig

Tutkinto
Tradenomi

Raportin/Opinnäytetyön nimi
REST-rajapinnan toteutus Flask-sovelluskehyksellä

Sivu- ja liitesivumäärä
30 + 15

Ohjelmointirajapinta on sovellusten tai laitteiden tapa olla vuorovaikutuksessa keskenään tiettyjä määritelmiä ja protokollia hyödyntäen. Verkkorajapinnat ovat ohjelmointirajapintoja, joissa vuorovaikutus tapahtuu verkon yli.

REST on suosittu arkkitehtuurityyli ohjelmointirajapintojen kehityksessä. Se antaa rajoitteet rajapinnan arkkitehtuurisiin valintoihin, mutta ei puutu lainkaan tekniseen toteutukseen. REST-tyylin rajoitteita noudattamalla rajapinnasta saadaan skaalautuva, helposti ylläpidettävä sekä selkeä.

Flask on yleisesti käytetty Python-ohjelmointikielien sovelluskehys verkkosovellusten kehittämiseen. Se tarjoaa minimaalisen kehyksen, ja antaa ohjelmoijalle paljon vapauksia ja myös vastuuta.

Tässä opinnäytetyössä oli tarkoitus tutkia, miten Flask soveltuu REST-tyyliä noudattavan rajapinnan toteuttamiseen. Työn tietoperustassa esiteltiin ensin ohjelmointirajapintoja ja erityisesti verkkorajapintoja. Toiseksi esiteltiin REST-tyylin taustaa ja sen asettamia rajoitteita rajapintojen kehitykseen. Tietoperustan lopuksi käsiteltiin HTTP-protokollaa, joka on hyvin käytetty tiedonsiirtoprotokolla verkkorajapintojen kehityksessä.

Työssä toteutettiin rajapinta kuvitteellisen blogisovelluksen tarpeisiin, ja toteutuksessa pyrittiin noudattamaan REST-tyylin asettamia rajoitteita. Työssä tarkasteltiin, miten REST-tyylin rajoitteita noudattavan rajapinnan kehittäminen onnistuu Flask-kehyksellä.

Toteutettua rajapintaa arvioidessa todettiin, että Flask soveltuu hyvin varsinkin yksinkertaisten REST-rajapintojen kehitykseen. Flask ei kuitenkaan ole nimenomaisesti REST-tyylin rajoitteita noudattavien rajapintojen kehitykseen tarkoitettu kehys, joten joihinkin rajoitteisiin vastaaminen on hieman hankalaa ja työlästä Flask-kehyksellä. Laajemman ja monimutkaisemman rajapinnan kehityksessä kannattaakin hyödyntää Flask-kehykseen liitettäviä kirjastoja ja lisäosia.

Asiasanat
REST, ohjelmointirajapinta, Flask, API, Python

Sisällys

1 Johdanto 1
2 Verkkorajapinnat 3
2.1 Ohjelmointirajapinta 3
2.2 Verkkorajapinta 4
2.3 Dokumentaatio 4
3 REST-arkkitehtuurityyli 6
3.1 REST-tyylin tausta 6
3.2 REST-rajapinnan toiminta 7
3.3 REST-tyylin arkkitehtuuriset elementit 8
3.4 REST-tyylin rajoitteet 10
3.5 REST-rajapintojen luokittelu 13
4 HTTP 15
4.1 Pyyntö ja vastaus 15
4.2 Metodit 16
4.3 Vastauskoodit 17
5 Työskentelymenetelmät 19
6 REST-rajapinnan toteutus Flask-sovelluskehyksellä 21
6.1 Toteutettava rajapinta 21
6.2 Flask ja REST-tyylin rajoitteet 21
6.2.1 Tilaton palvelin 22
6.2.2 Yhtenäinen rajapinta 22
6.2.3 Itseään kuvaavat viestit 23
6.2.4 Välimuisti 24
6.2.5 Kerroksellinen järjestelmä ja ladattava koodi 24
6.2.6 Dokumentaatio 24
7 Yhteenveto 26
Lähteet 28
Liitteet 31
Liite 1. Resurssin esitys JSON-muodossa 31
Liite 2. Hypertekstiä sisältävä resurssin esitys 32
Liite 3. HTTP-pyyntö (Richardson & Ruby 2007, alaluku HTTP: Documents in Envelopes) 33
Liite 4. HTTP-vastaus (Richardson & Ruby 2007, alaluku HTTP: Documents in Envelopes) 34
Liite 5. Rajapinnan toiminnot lisätietoineen 35
Liite 6. Flask-kehyksellä tehty minimalistinen rajapinta 36
Liite 7. Flaskin blueprint-objektin routet, eli päätepisteet 37

Liite 8. GET-pyynnön vastaus JSON-muodossa 38
Liite 9. Yhden blogipostaus-resurssin palauttavan päätepisteen toteutus 39
Liite 10. Blogipostauksen tietokantaluokan resurssin esityksen palauttava osa 40
Liite 11. Välimuistituksen sallivan tiedon lisääminen HTTP-vastauksiin 42
Liite 12. Swagger-dokumentaation alustaminen 43
Liite 13. Osa Swagger-dokumentaatiosta YAML-muodossa 44

Käsitteet

API	Application Programming Interface, ohjelmointirajapinta ohjelmistojen välistä vuorovaikutusta varten
CRUD	Create, Read, Update and Delete, neljä perustoimintoa tiedon varastointiin
DevOps	Jatkuvan integraation ja kehityksen automatisointiin liittyvät toimet ohjelmistokehityksessä
Flask	Avoimen lähdekoodin verkkosovelluskehys, joka on kirjoitettu Pythonilla
HTML	Hypertext Markup Language, verkkosivujen rakenteen ja sisällön määrittelevä merkintäkieli
HTTP	Hypertext Transfer Protocol, tiedonsiirtoon tarkoitettu verkkoprotokolla
Hypermedia	Tietomuoto, joka sisältää linkkejä multimediaan, kuten kuviin tai videoihin
Hyperteksti	Linkkejä muihin asiaan liittyviin tietoihin tai dokumentteihin sisältävää tekstiä
JSON	JavaScript Object Notation, avain-arvo-pareihin perustuva tiedostotyyppi. Suosittu erityisesti verkkoprotokollien yhteydessä.
Python	Yksi käytetyimmistä ohjelmointikielistä
REST	Representational State Transfer, arkkitehtuurityyli ohjelmointirajapinnoille ja hypermediajärjestelmille
Swagger	Työkalu rajapintojen dokumentaation luomiseen ja hallintaan
URI	Resurssien tunnistamiseen käytettävä verkko-osoite
WSGI	Web Server Gateway Interface, Pythonin rajapinta web-palvelimen ja sovelluskehyksen välillä

YAML	YAML Ain’t Markup Language, tietojen sarjoitukseen käytettävä dataformaatti

1 Johdanto

Ohjelmointirajapinnoilla on jo pitkä historia, mutta verkkorajapinnat alkoivat yleistyä vasta 2000-luvun alussa. Tänä päivänä ohjelmointirajapinnasta puhuttaessa tarkoitetaan yleensä verkkorajapintaa (engl. Web API), ja niiden yleistyminen muutti kokonaan käsityksemme verkossa tapahtuvasta liiketoiminnasta. Tänä päivänä lähes jokaisen verkossa toimivan palvelun taustalla toimii verkkorajapinta. (Lane 10.10.2019)

Rajapintojen kehityksessä voidaan hyödyntää montaa erilaista tyyliä ja tekniikkaa. Arkkitehtuurisesta näkökulmasta REST on ylivoimaisesti suosituin tyyli verkkorajapintojen kehityksessä. (Postman 2022.) Tässä opinnäytetyössä avataan REST-arkkitehtuurityyliä verkkorajapintojen toteutuksen yhteydessä, ja työn toteutusvaiheessa toteutetaan REST-rajapinta teoriaosuuden oppeja hyödyntäen.

Tämän työn tavoite on selvittää minkälaisia arkkitehtuurisia rajoitteita REST-tyyli asettaa, miten noita rajoitteita noudattava rajapinta toteutetaan sekä mitä hyötyjä REST-tyylin noudattaminen tuo rajapintojen toteuttamiseen. Toinen selvitettävä asia on Flask-sovelluskehyksen soveltuvuus REST-rajapinnan toteuttamiseen. Haluan selvittää, onko REST-tyylin rajoitteisiin vastaavan verkkorajapinnan toteuttaminen mahdollista Flask-sovelluskehystä hyödyntäen. Kolmas tavoite on oppia lisää REST-rajapinnoista sekä ohjelmoinnista käyttäen Pythonia ja Flask-sovelluskehystä. Olen aikaisemminkin toteuttanut rajapintoja, mutta en koskaan ole tutustunut tarkemmin REST-tyylin periaatteisiin ja rajoitteisiin. Myös Python ja Flask ovat itselleni pintapuolisesti tuttuja, mutta en ole käyttänyt kumpaakaan laajojen projektien kehityksessä, joten toivon oppivani paljon lisää molemmista.

Työn toisessa luvussa käsitellään ohjelmointirajapintoja yleisesti sekä verkkorajapintoja. Kolmannessa luvussa syvennytään REST-arkkitehtuurityyliin ja sen rajoituksiin, jotka ohjaavat REST-rajapintojen kehitystä. Neljännessä luvussa käsitellään HTTP-protokollaa, joka on keskeinen osa verkkorajapintojen toimintaa. Luvussa käydään läpi vastaus-pyyntö-protokolla, HTTP-metodit sekä vastauskoodit. Viidennessä luvussa käydään lyhyesti läpi toteutuksen työskentelymenetelmät, ja kuudennessa luvussa suunnitellaan ja toteutetaan REST-tyylin rajoitteisiin vastaava rajapinta Pythonin Flask-sovelluskehystä käyttäen. Viimeisessä luvussa arvioidaan toteutettua rajapintaa REST-tyylin rajoitteiden näkökulmasta ja Flask-kehystä REST-rajapinnan toteuttamisessa.

Rajapinnan toteuttamista käsittelevässä luvussa käydään läpi vain REST-tyylin kannalta olennaiset asiat. Siitä on siis rajattu pois yleisesti Flask-sovelluskehyksen käyttöön ja toimintaan liittyvät seikat. Tätä opinnäytetyötä ei voi siis käyttää aloittelijan opiskelu- tai ohjemateriaalina Flask-

kehyksen käyttöön, vaan työstä saatava hyöty on nimenomaan REST-tyyliä noudattavan rajapinnan kehitykseen liittyvät seikat Flask-kehyksessä.

2 Verkkorajapinnat

Ohjelmointirajapintojen historia ulottuu 1960-luvulle, kun ohjelmistokehittäjät alkoivat jakaa rakentamiaan ohjelmistokirjastoja toisten kehittäjien käytettäväksi. Toiset kehittäjät pystyivät käyttämään näitä kirjastoja ilman, että tunsivat niiden sisältämää koodia. 70- ja 80-luvulla, verkkoon kytkettyjen tietokoneiden ilmaantuessa, syntyivät ensimmäiset verkkorajapinnat, joita pystyttiin käyttämään verkon välityksellä. Internetin käytön yleistyessä alkoivat myös verkkorajapinnat yleistyä, ja yhä useampi yritys alkoi tarjoamaan hyödyllisiä palveluja verkkorajapintojen kautta. (Jin, Sahni & Shevat 2018, alaluku Preface.)

Tässä luvussa käsitellään ohjelmointirajapintoja yleisellä tasolla, sekä perehdytään erityisesti verkkorajapintoihin.

2.1 Ohjelmointirajapinta

Ohjelmointirajapinta eli API (Application Programming Interface) on joukko määritelmiä ja protokollia ohjelmistojen rakentamiseen ja integroimiseen. Ohjelmistorajapinta mahdollistaa järjestelmien kommunikoinnin toisten järjestelmien kanssa tarvitsematta tietoa siitä, miten ne on toteutettu. API voidaan nähdä eräänlaisena sopimuksena eri osapuolten välillä, jossa määritellään miten osapuoli vastaa toisen osapuolen sille lähettämään pyyntöön. (Red Hat 2022.)

Ohjelmointirajapintoja on monenlaisten eri järjestelmien, kuten käyttöjärjestelmien tai ohjelmointikirjastojen käyttötarkoituksiin. Käyttöjärjestelmissä rajapinta auttaa sovellusta kommunikoimaan järjestelmän alempien kerrosten kanssa sekä muiden sovellusten kanssa noudattamalla protokollien ja määritelmien joukkoa. Yksi esimerkki tällaisesta määrittelystä on POSIX (Portable Operating System Interface), jonka määritelmiä seuraamalla tietylle käyttöjärjestelmälle toteutettu sovellus toimii myös muissa samaa POSIX-määrittelyä noudattavissa käyttöjärjestelmissä. (Pedro 2017.)

Ohjelmistokirjastot ovat tärkeässä roolissa, kun halutaan luoda monien järjestelmien kanssa yhteensopivia sovelluksia. Kirjastoja käyttävien sovellusten tulee noudattaa sääntöjä, jotka kirjaston rajapinta määrittää. Kirjastojen saman rajapinnan noudattaminen mahdollistaa ohjelmistokehittäjille monien eri kirjastojen kanssa kommunikoivien sovellusten kehittämisen. Tämä mahdollistaa myös saman ohjelmointikirjaston hyödyntämisen useilla eri ohjelmointikielillä. (Pedro 2017.)

2.2 Verkkorajapinta

Verkkorajapinta (engl. Web API) on ohjelmointirajapinta, jonka erityispiirteenä on kommunikoinnin ja tiedonsiirron tapahtuminen verkon ja erilaisten internet-protokollien avulla. Verkkorajapinta määrittelee joukon päätepisteitä sekä pyyntö- ja vastausrakenteet, joiden avulla rajapintaa käytetään. Myös vastausten tuetut mediatyypit ovat yleensä identifioitu. Yleisimpiä vastausten mediatyyppejä ovat JSON ja XML. (Pedro 2017.)

Tänä päivänä lähes jokainen verkkosivusto ja -sovellus käyttää verkkorajapintaa tiedon, kuten sisällön ja median hakemiseen. Myös mobiilisovellukset hyödyntävät verkkorajapintoja reaaliaikaisen tiedon hakuun. (Lane 26.9.2022)

Rajapintoja on hyvin moniin eri tarkoituksiin, mutta karkeasti ne voidaan jakaa neljään eri ryhmään: avoimiin rajapintoihin, kumppanirajapintoihin, sisäisiin rajapintoihin sekä yhdistelmärajapintoihin. Avoimet rajapinnat ovat nimensä mukaisesti avoinna yleisesti ohjelmistokehittäjille ja muille käyttäjille minimaalisilla rajoituksilla. (Wilfred 3.9.2022) Julkiset rajapinnat vaativat usein rekisteröitymisen, ja jotkut yritykset veloittavat maksun rajapinnalle lähetetyistä kutsuista. (Bigelow 8.2.2021)

Kumppanirajapinnat ovat yrityksen tarjoama rajapinta sen kumppaniyritykselle. Ne eivät ole julkisesti saatavilla, vaan niiden käyttö vaatii erityisen oikeuden. (Wilfred 3.9.2022) Kumppanirajapinnat ovat julkisia rajapintoja tiukempia siitä, kenelle tarjoavat palveluitaan. Ne voivat olla maksullisia tai ilmaisia. Koska ne on tarkoitettu vain tiettyjen tahojen käyttöön, on niissä tiukat säännöt tunnistautumista ja valtuutusta varten. (Simpson 15.3.2022)

Sisäinen rajapinta on tarkoitettu vain yrityksen sisäiseen käyttöön yhdistämään järjestelmiä ja tietoa yrityksen sisällä. Sisäinen rajapinta voi esimerkiksi yhdistää organisaation palkka- ja HR-järjestelmät. (Bigelow 8.2.2021)

Yhdistelmärajapinnat yhdistävät monta eri rajapintaa ja mahdollistavat monen eri palvelun hyödyntämisen yhdellä API-kutsulla (Wilfred 3.9.2022). Yhdistelmärajapinta voi olla nopeampi ja tehokkaampi ratkaisu yksittäisen rajapinnan sijaan (Bigelow 8.2.2021).

2.3 Dokumentaatio

API-dokumentaation on käsikirja, joka kertoo käyttäjille, yleensä kehittäjille, miten rajapintaa käytetään. Dokumentaation suunnittelu, kattavuus ja helppokäyttöisyys ovat tärkeitä asioita, kun halutaan varmistaa, että kehittäjät saavat hyvän käyttökokemuksen rajapinnan käytöstä. Laadukas

dokumentaatio auttaa myös lyhentämään uusien käyttäjien käyttöönottoaikaa ja vähentää tarvetta ottaa yhteyttä asiakaspalveluun apua varten.

Hyvä dokumentaatio sisältää useita elementtejä, kuten aloitusoppaan, opetusohjelmia ja interaktiivista dokumentaatiota, jotta kehittäjät voivat kokeilla API-kutsuja. Dokumentaatiossa tulisi olla esimerkkejä jokaisesta kutsusta, jokaisesta parametrista ja vastauksista jokaiselle kutsulle. Sen tulisi myös sisältää koodiesimerkkejä yleisesti käytetyistä ohjelmointikielistä. Dokumentaatiossa pitäisi selittää jokainen API-kutsu ja siinä tulisi olla myös esimerkkejä virheilmoituksista. Tärkeää on myös pitää dokumentaatio ajan tasalla. (Wagner s.a.)

3 REST-arkkitehtuurityyli

Rajapinnan kautta liikutellaan komentoja ja tietoja, ja tämä vaatii selkeää protokollaa ja arkkitehtuuria, eli sääntöjä, rakenteita ja rajoituksia, jotka ohjaavat rajapinnan toimintaa. Arkkitehtuurisia ratkaisuja on useita, ja jokainen niistä soveltuu tiettyyn tarkoitukseen. (Bigelow 8.2.2021) Rajapintojen toteutuksessa REST on tänä päivänä suosituin arkkitehtuurityyli (Postman 2022).

REST (Representational State Transfer) on arkkitehtuurityyli hajautetuille hypermediajärjestelmille, kuten sen ilmaisi Roy T. Fielding väitöskirjassaan (Fielding 2000, 76). REST nojautuu asiakas-palvelin-malliin, joka erottaa järjestelmän osapuolet toisistaan ja tarjoaa näin joustavuutta kehityksessä ja toteutuksessa (Bigelow 8.2.2021).

REST ei ole tarkkoja sääntöjä rajapinnan toteutukseen antava protokolla, vaan joukko arkkitehtuurisia periaatteita ja ohjeita rajapinnan rakentamiseen. Se ei ota kantaa rajapinnan tekniseen toteutukseen, vaan jättää ne ohjelmistokehittäjien päätettäviksi. (Red Hat 2019.)

Tässä luvussa käsitellään REST-arkkitehtuurityyliä, ja miten toteutetaan RESTful API, eli REST-tyylin periaatteita noudattava ohjelmointirajapinta.

3.1 REST-tyylin tausta

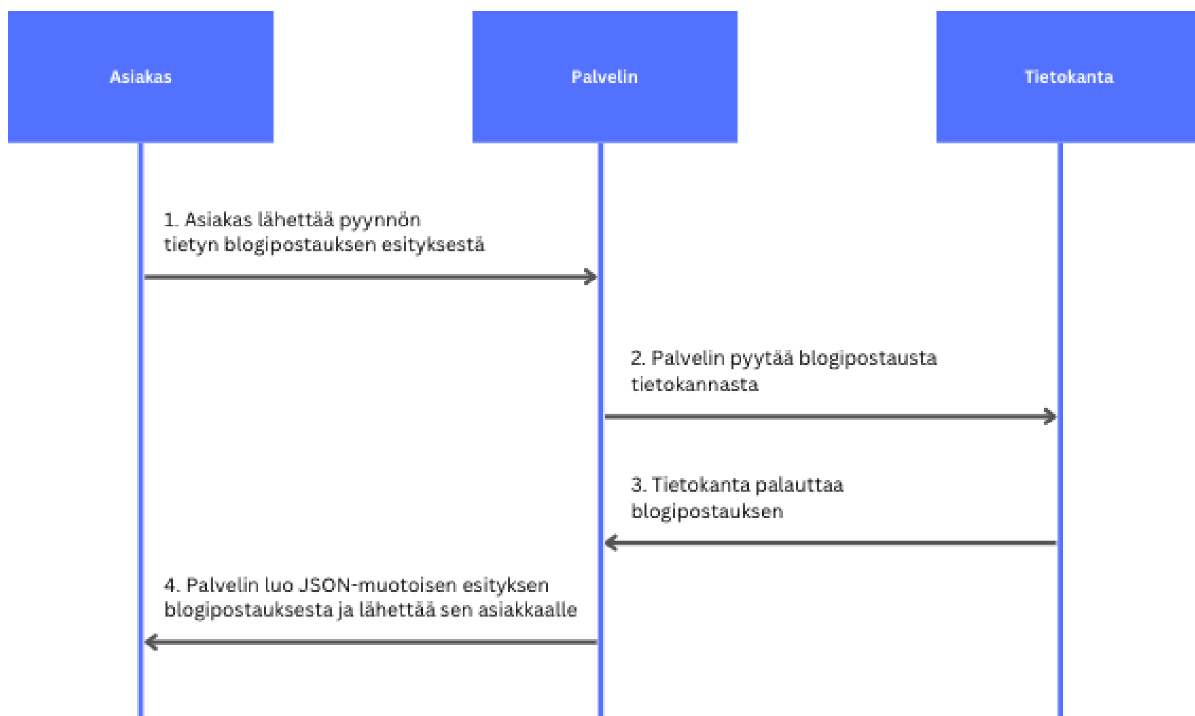
Ensimmäinen versio REST-mallista kehitettiin vuosina 1994–1995, kun Fielding työskenteli HTTP-protokollan parissa. REST-tyyliä käytettiin tunnistamaan varhaisten HTTP-versioiden ongelmia modernin verkkoarkkitehtuurin tukemisessa. (Fielding 2000, 109–116.)

Alun perin REST-tyyliä kutsuttiin nimellä HTTP object model. Se olisi kuitenkin saattanut aiheuttaa väärinkäsityksiä, joten nimi vaihdettiin. Nimen Representational State Transfer on tarkoitus herättää mielikuva siitä, miten hyvin suunniteltu verkkosovellus toimii; ”a network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.” (Fielding 2000, 109–116.)

REST-tyyliä käytettiin onnistuneesti verkkostandardien, kuten HTTP1.1 ja URI-syntaksin kehityksessä. Molemmat ovat kestäneet hyvin aikaa, huolimatta niiden keskeisestä roolista internetprotokollina ja laajasta käytöstä. Voidaan siis ajatella, että REST-tyyliä ja sen rajoitteita seuraamalla voidaan luoda kestäviä ja skaalautuvia järjestelmiä. (Lange 2016, 2.)

3.2 REST-rajapinnan toiminta

REST on asiakas-palvelin-kommunikaatioon pohjautuva arkkitehtuurityyli, jossa asiakas lähettää pyynnön palvelimelle, ja palvelin vastaa tähän pyyntöön. REST-tyylissä asiakas pyytää palvelimelta jonkin resurssin tilaa, ja palvelin palauttaa tuon resurssin tilan esityksen. Resurssi voi olla esimerkiksi tietokantaan tallennettu blogipostaus, ja palvelimen palauttama esitys sisältää yksinkertaisen listan tämän tietokantatietueen arvoista. (Kuva 1.)



Kuva 1. REST-tyylin toimintaperiaate (mukaillen Lange 2016, 3)

REST-järjestelmissä yleisesti käytetty resurssien esitysmuoto on JSON (JavaScript Object Notation), joka on yksinkertainen nimi-arvo-pareihin perustuva tiedostomuoto. JSON on suosittu tiedostomuoto, koska kaikki ohjelmointikielet ymmärtävät sitä, se on ihmisen ja koneen luettavissa sekä se on kevyt (Red Hat 2019). Liitteessä 1 on kuvattu, miltä kuvassa 1 haettavan blogipostauksen esitys voisi JSON-muodossa esimerkiksi näyttää.

Kun asiakas on vastaanottanut resurssin esityksen, voi se muokata sitä, esimerkiksi vaihtaa otsikon, ja lähettää päivitetyn esityksen takaisin palvelimelle. Palvelin päivittää sitten oman resurssinsa tietokantaan päivitetystä esityksestä saamillaan tiedoilla. REST tarkoittaa siis käytännössä resurssien esitysten siirtämistä asiakkaan ja palvelimen välillä. (Lange 2016, 3–4.)

3.3 REST-tyylin arkkitehtuuriset elementit

REST-tyylissä on määritelty kolme arkkitehtuurista elementtiä, jotka ohjaavat resurssien hallintaa. Näihin elementteihin lukeutuvat komponentit, liittimet ja dataelementit. Lisäksi REST sisältää kuusi rajoitetta, jotka vaikuttavat siihen, miten komponentit, liittimet ja dataelementit toimivat yhdessä resurssien hallinnassa.

Komponentit ovat erilaisia ohjelmistoja, jotka toimivat yhteydessä toisiinsa. Ne toimivat liittimen kautta yhdistettyinä päätepisteinä, jotka vaihtavat keskenään dataelementtejä. Komponentit on jaoteltu neljään eri tyyppiin, jotka ovat alkuperäpalvelin, käyttäjäagentti, yhdyskäytävä sekä välityspalvelin.

Alkuperäpalvelin (Origin Server) on suunniteltu vastaanottamaan pyyntöjä, jotka muokkaavat resurssien arvoja, ja se toimii ikään kuin säiliönä, joka sisältää pyydettyjen resurssien esitykset. Se käyttää palvelinliitintä vastaanottaakseen ja vastatakeen pyyntöihin. Verkkopalvelimet, kuten Apache Tomcat ja Microsoft IIS ovat esimerkkejä alkuperäpalvelimista.

Käyttäjäagentti (User Agent) käynnistää pyynnön asiakkaan päässä. Se käyttää asiakasliitintä näiden pyyntöjen aloittamiseksi. Se on myös palvelimelta tulevan vastauksen vastaanottaja. Verkkoselaimet, kuten Chrome ja Mozilla, ovat tyypillisiä esimerkkejä käyttäjäagenteista, jotka toimivat alkupisteenä käyttäjän pyynnölle ja renderöivät vastauksen palvelimelta käyttäjälle nähtävään muotoon.

Yhdyskäytävä (gateway) on verkon välityksellä toimiva välittäjä, joka tarjoaa palvelukapseloinnin, kuten datan kääntämisen, suorituskyvyn parantamisen tai turvallisuuden vahvistamisen salaamisen kautta. Se toimii sekä asiakkaana että palvelimena helpottaakseen pyyntöjen ja vastausten vaihtoa mahdollisen datan kääntämisen kera. Esimerkkejä yhdyskäytävistä ovat Squid, NGINX ja CGI.

Myös välityspalvelin (proxy) toimii välittäjän asiakkaan tekemille pyynnöille jostakin palvelusta tai resurssista. Se tarjoaa käyttöliittymän palveluiden kapselointiin, kuten kuormatasapainotukseen, salaamiseen tai pakkaamiseen. Välityspalvelinkomponentteina REST-tyylisessä järjestelmässä voivat olla esimerkiksi CERN-proxy, Netscape-proxy tai CGI-proxy.

Liittimet hallinnoivat kommunikointia eri komponenttien välillä. REST-tyylissä on viisi erilaista liitintyyppiä, jotka kapseloivat resurssien käyttötoiminnot ja mahdollistavat resurssin esitysten siirtelyn. Kaikki liitinten kautta tapahtuva kommunikointi on piilotettua. Liitinten viisi eri tyyppiä ovat asiakas, palvelin, välimuisti, tulkki sekä tunneli.

Asiakas aloittaa kommunikoinnin lähettämällä pyynnön ja vastaanottaa palvelimen vastauksen. Palvelin kuuntelee yhteyksiä ja antaa vastaukset pyyntöihin mahdollistan pääsyn sen palveluihin. Välimuisti on kolmas liitintyyppi, joka tallentaa uudelleenkäytettäviä vastauksia samankaltaisiin vuorovaikutuksiin. Esimerkkejä tästä liitintyypistä ovat verkkoselaimen välimuisti tai Microsoft Azure CDN.

Tulkki kääntää resurssin tunnisteen verkon osoitteeksi. Useimmat URI:t sisältävät DNS-hostin. Verkkoselain poimii tuon hostin ja käyttää DNS-tulkkia resurssin nimeämiselimen IP-osoitteen hakemiseen. Tunneli välittää tietoa yhteyden rajojen, kuten palomuurien tai yhdyskäytävien, yli. REST-komponentit voivat vaihtaa aktiivisesta tunnelikäyttäytymiseen, mikä mahdollistaa asiakkaan suoran vuorovaikutuksen etäpalvelimen kanssa käyttämällä protokollaa, kuten TLS. (Kaushik 2022.)

REST-tyylissä määritellään kuusi dataelementtiä, jotka ovat resurssi, resurssin tunniste, resurssin metadata, esitys, esityksen metadata sekä ohjaustiedot. Resurssi on hyvin keskeinen käsite REST-tyylissä. REST-mallissa kaikkea nimettävää tietoa kutsutaan resursseiksi. Resurssi voi olla esimerkiksi dokumentti, kuva tai muiden resurssien kokoelma. (Fielding 2000, 88.)

Resurssien tunnistamiseen käytetään URI-osoitetta (Uniform Resource Identifier). URI sisältää resurssin nimen ja osoitteen. Kaikki palvelimen tarjoama tieto voidaan identifioida resursseiksi. Esimerkiksi URI ”http://www.api.foo.com/asiakkaat” identifioi resurssin nimeltään asiakkaat. Halutessaan käsitellä resurssia, asiakas muodostaa yhteyden URI-osoitteessa eriteltyyn palvelimen osoitteeseen (esimerkin tapauksessa www.api.foo.com) ja sen resurssin polkuun (/asiakkaat). Esimerkin URI-osoitteeseen lähetetty GET-metodi palauttaisi asiakas-resurssikokoelman. (De 2017, alaluku Designing a RESTful API.)

Jokaisella resurssilla tulee olla oma uniikki URI, jolla se voidaan tunnistaa. URI tulisi olla suunniteltu niin, että se ei muutu, vaikka sen identifioima resurssi päivitettäisiinkin. Hyvä käytäntö URI-osoitteiden määrittelyssä on käyttää resurssikokoelmien osoitteessa monikkomuotoa, esimerkiksi /asiakkaat. Yksittäisen resurssiin kokoelman sisällä päästään käsiksi esimerkiksi lisäämällä URI-osoitteen perään resurssin uniikki id (/asiakkaat/1). (Lange 2016, 8–9.) Resurssin metadata kuvaa resurssin eri ominaisuuksia antamalla lisätietoja, kuten resurssin sijainnin tai vaihtoehtoisen resurssin tunnisteen. Metadata luokittelee resurssin ja antaa siitä tietoa. (Kaushik 2022.)

REST-tyylissä asiakas ei ole suorassa vuorovaikutuksessa palvelimen resurssien kanssa, vaan palvelin palauttaa asiakkaalle resurssin tilan. Tätä ominaisuutta kutsutaan resurssien manipuloinniksi esitysten kautta. Se tarkoittaa, että asiakas ei voi esimerkiksi suoraan ajaa

tietokantakomentoja tietokannan tauluihin, vaan asiakkaalle näytetään resurssin tila esimerkiksi JSON-muodossa. Jos asiakas haluaa päivittää resurssin tilan, sille lähetetään resurssin esitys, jota asiakas päivittää uusilla tiedoilla. Tämä päivitetty tilan esitys lähetetään takaisin palvelimelle ja pyydetään palvelinta päivittämään resurssiaan niin, että se vastaa päivitettyä esitystä. (Lange 2016, 11–12.) Resurssin esitys koostuu varsinaisesta datasta, dataa kuvaavasta metadatasta sekä hypermedialinkeistä, jotka auttavat asiakasta siirtymään seuraavaan haluttuun tilaan (Gupta 7.4.2022.)

Sama resurssi voidaan esitellä eri asiakkaille eri tavoin. Esimerkiksi käyttöliittymäasiakkaalle voidaan palauttaa resurssi HTML-muodossa, sovellusasiakkaalle taas JSON-muodossa. Resurssin esitys on asiakkaalle tapa olla vuorovaikutuksessa resurssin kanssa, mutta se ei kuitenkaan ole itse resurssi. (De 2017, alaluku Designing a RESTful API.)

Esityksen metadata kuvailee vastauksen rungon muototyyppiä. Tämä auttaa tulkitsemaan dataa asiakkaan päässä. Esityksen metadata koostuu pyynnössä ja vastauksessa käytetyistä eri otsikoista. Näitä otsikoita voivat olla esimerkiksi content-type ja content-length eli vastauksen sisällön tyyppi ja maksimipituus.

Ohjaustiedot määrittelevät asiakkaan ja palvelimen välisten viestien tarkoituksen, kuten pyydetyn toiminnon ja vastauksen merkityksen. Ne myös auttavat parametrisoimaan pyyntöjä ja ohittamaan yhteyden muodostavan elementin oletuskäyttäytymisen. Ohjaustiedot voivat esimerkiksi muuttaa pyynnön tai vastauksen välimuistin käyttäytymistä. (Kaushik 2022.)

3.4 REST-tyylin rajoitteet

Fielding (2000, 78–85) esittelee väitöskirjassaan kuusi rajoitetta REST-malliin pohjautuville järjestelmille. Rajoitteiden tarkoitus on varmistaa, että järjestelmän arkkitehtuuri tukee järjestelmän tarvitsemia toiminnallisuuksia ja ominaisuuksia. Rajoitukset myös edistävät arkkitehtuurin ylläpidettävyyttä, joustavuutta sekä laajennettavuutta. (Fielding 2000, 13.) Fieldingin määrittämää kuutta rajoitetta noudattamalla toteutetun järjestelmän voidaan sanoa olevan niin kutsuttu RESTful järjestelmä (Lange 2016).

Kuten aiemmin jo todettiin, REST-rajapinnan tulee noudattaa asiakas-palvelin-mallia, jossa järjestelmän eri komponentit on erotettu toisistaan. Näin järjestelmän molempia puolia voidaan kehittää riippumattomina toisistaan. Kun esimerkiksi tietokannan toiminnot ovat eroteltu käyttöliittymän toiminnoista, helpottaa se käyttöliittymän muokattavuutta useille eri alustoille. Myös skaalautuvuus paranee, kun järjestelmän komponentit ovat yksinkertaisempia. (Fielding 2000, 78.)

Toinen rajoitus REST-rajapinnalle on asiakas-palvelin-kommunikoinnin tilattomuus. Käyttöliittymän lähettämän pyynnön palvelimelle tulee sisältää kaikki pyynnön toteuttamiseen tarvittava tieto. Istunnon tila tulee siis pitää kokonaan asiakkaalla. Tämä rajoitus parantaa järjestelmän luotettavuutta ja skaalautuvuutta, sillä palvelin voi vapauttaa nopeasti resurssejaan uusiokäyttöön, kun sen ei tarvitse välittää asiakasrajapinnan tilasta. (Fielding 2000, 77–79.)

Jos jonkin resurssin tila täytyy säilyttää useamman kuin yhden pyynnön ajan, on asiakkaan tehtävä huolehtia tilan tietojen uudelleenlähettämisestä. Tämä poistaa palvelimelta vaatimuksen tilan ylläpitämisestä ja päivittämisestä, ja auttaa palvelimen skaalautuvuudessa. (De 2017, alaluku REST Principles.) Toisaalta tilaton palvelin voi myös heikentää suorituskykyä, kun samaa dataa joudutaan lähettämään toistuvasti eri pyynnöissä eikä sitä voida säilyttää palvelimella. (Fielding 2000, 79.)

Fielding määrittelee yhdeksi REST-tyylin rajoituksesi myös pyrkimyksen yhtenäiseen rajapintaan eri komponenttien välillä. Yhtenäistämällä komponenttirajapintaa kokonaisjärjestelmän arkkitehtuuri on yksinkertaisempi ja sen toimintojen näkyvyys paranee. (Fielding 2000, 81–82.) REST erottuu muista arkkitehtuurityyleistä varsinkin tällä yhtenäisen rajapinnan rajoituksella (Lange 2016, 7).

Jotta yhtenäinen rajapinta saavutetaan, on järjestelmän eri komponenttien käyttäytymistä ohjattava arkkitehtuurisilla rajoitteilla. Nämä neljä rajoitetta ovat resurssien tunnistaminen, resurssien manipulointi esitysten avulla, itseään kuvaavat viestit sekä hypermedia sovelluksen tilan moottorina. (Fielding 2000, 82.) Edellisessä luvussa käsiteltiin jo resurssien tunnistamista sekä resurssien manipulointia esitysten kautta, joten seuraavaksi käydään läpi itseään kuvaaviin viesteihin sekä hypermediaan liittyvät rajoitteet.

Itseään kuvaavien viestin rajoitteella pyritään siihen, että jokaisen pyynnön ja vastauksen tulee sisältää tarpeeksi tietoa, jotta vastaanottaja tietää, kuinka viesti tulee prosessoida. Jokaisella viestillä tulisi olla eriteltynä esimerkiksi mediatyyppi, joka kertoo vastaanottajalle, kuinka viesti tulee jäsentää. (Lange 2016, 12.)

Jos järjestelmä käyttää tiedonvälitykseen HTTP-protokollaa, tulee HTTP-metodien muodollista merkitystä noudattaa, jotta asiakas ymmärtäisi metodien käyttötarkoituksen ilman muita tietoja. Selkeän URI-määrityksen ja HTTP-metodien tarkoituksenmukaisen käytön etuna on, että asiakas voi ilman muuta tietoa ymmärtää, miten järjestelmä toimii. (Lange 2016, 12–14.) HTTP-metodeja käsitellään lisää luvussa 4.3.

Hypermedia on Ted Nelsonin vuonna 1965 esittelemä termi, jolla tarkoitetaan kuvia, videoita, ääntä, tekstiä ja linkkejä sisältävää sisältöä. Se on askeleen pidemmälle viety termi hypertekstistä,

joka tarkoittaa linkkejä toisiin kohteisiin sisältävää tekstiä. Esimerkki jokapäiväisessä käytössämme olevista hyperteksteistä ja hypermediasta on internetsivustot. Internetsivustojen ulkoasu on kirjoitettu HTML-merkintäkielellä, jota internetselaimet tulkitsevat, ja sivustot sisältävät linkkejä ja muuta interaktiivista mediaa, joka ohjaa käyttäjän toimintaa sivustolla. (Stowe 2014.)

REST-tyylissä hypermedialla on keskeinen rooli, sillä onhan REST alun perin arkkitehtuurityyli hajautetuille hypermediajärjestelmille (Fielding 2000, 76). REST-tyylin rajoite hypermedia sovellustilan moottorina tarkoittaa, että palvelin ohjaa asiakkaan toimintaa hypermedian avulla. (Richardson, Amundsen & Ruby 2013, luku 4.) Tässä kontekstissa sovellus tarkoittaa palvelimella pyörivää verkkosovellusta (Nadkarni 2022). Käytännössä tämä ilmenee siten, että asiakkaalle lähetettävä resurssi sisältää siihen liittyviä linkkejä. Nämä linkit tarjoavat tietoa jatkotoimenpiteistä ja kuinka selata muita resursseja tarkoituksenmukaisella tavalla. Esimerkiksi pyydettäessä tietoa pankkitilistä, voi vastaus sisältää linkkejä toimintoihin kuten talletus, nosto tai siirto. (De 2017, ala-luku Hypermedia as the Engine of Application State (HATEOAS).) Liitteessä 2 on esimerkki hypertekstiä sisältävästä resurssin esityksestä.

Rajoite hypermediasta sovellustilan moottorina on kenties eniten väärinymmärretty ja -käytetty kaikista REST-tyylin rajoitteista. Fielding kirjoitti blogiinsa aihetta käsittelevän kirjoituksen, jossa hän painotti, että jos sovellustilan moottorina ei ole hyperteksti, ei sitä voida kutsua REST-rajapinnaksi. (Fielding 20.10.2008.)

Jos järjestelmässä toteutuu nämä neljä edellä kuvattua osarajoitetta, täyttää se myös yhtenäisen rajapinnan rajoitteen. Yhtenäisyyden vaatimuksella on myös huonot puolensa. Se saattaa heikentää tehokkuutta, koska tietoa liikutellaan standardoidussa muodossa eikä välttämättä asiakkaan tarpeet huomioonottavassa muodossa. (Fielding 2000, 81–82.)

REST-tyylin välimuistia koskevan rajoitteen mukaan jokaisen vastauksen kohdalla tulee määritellä, onko se tallennettavissa välimuistiin vai ei. Välimuistiin tallennettujen vastausten kohdalla asiakas voi halutessaan käyttää tuota saman vastauksen tietoja mahdollisia myöhempiä vastaavia pyyntöjä varten. Tämä vähentää turhaa vuorovaikutusta asiakkaan ja palvelimen välillä, kun joitakin pyyntöjä ei tarvitse välittää palvelimelle asti. Se taas lisää järjestelmän tehokkuutta ja suorituskykyä. (Fielding 2000, 79–80.) Välimuistin käytöllä voi olla myös luotettavuutta heikentävä vaikutus, sillä joskus välimuistista haettu data voi olla vanhentunutta, ja erota siitä tiedosta, joka olisi saatu, jos pyyntö olisi lähetetty suoraan palvelimelle. (Fielding 2000, 80.)

Kerroksellisen järjestelmän rajoitus tarkoittaa, että järjestelmä on rakennettu hierarkkisista kerroksista ja jokainen kerros näkee ainoastaan sen kerroksen, jonka kanssa itse on vuorovaikutuksessa. (Fielding 2000, 82.) Kerroksellisessa järjestelmässä esimerkiksi asiakas

tuntee ainoastaan sen kerroksen, jonka kautta se on yhteydessä palvelimeen, mutta ei sen takana mahdollisesti olevia kerroksia. Jos palvelimen ja asiakkaan välille asetetaan välityspalvelin, asiakkaan ja palvelimen välinen kommunikointi ei muutu tai niiden koodia ei tarvitse päivittää. Kerroksellinen järjestelmä mahdollistaa myös palvelimen kommunikoinnin useisiin muihin palvelimiin muodostaessaan vastausta asiakkaalle. (Lange 2016, 17.)

Viimeinen REST-tyylin rajoitteista on ladattavan koodin rajoite. Se tarkoittaa, että asiakkaan toiminnallisuutta voidaan laajentaa palvelimelta ladattavan ja suoritettavan koodin, esimerkiksi skriptien, kautta (Fielding 2000, 84; Lange 2016, 18). Tämä yksinkertaistaa asiakaskomponenttia vähentämällä sille toteutettujen ominaisuuksien määrää. Toisaalta se myös heikentää järjestelmän näkyvyyttä, joten se onkin ainut valinnainen rajoite REST-tyylissä. (Fielding 2000, 84.)

Vaihtoehtoisen rajoitteen käsite voi kuulostaa hieman ristiriitaiselta, mutta sillä on tarkoituksensa järjestelmän arkkitehtuurisessa suunnittelussa, kun käsitellään useiden organisaatioiden rajapintoja. Vaihtoehtoisen rajoitteiden käyttö tarkoittaa, että arkkitehtuuri saa hyödyn vain silloin kun rajoitteiden vaikutus on tiedossa jonkin kokonaisjärjestelmän osalta. Esimerkiksi, jos organisaation kaikki asiakasohjelmat tukevat Java-sovelmia, organisaation sisäisiä palveluja voidaan rakentaa siten, että ne hyötyvät parannetusta toiminnallisuudesta lataamalla Java-luokkia. Samalla kuitenkin organisaation palomuuri voi estää Java-sovelmien siirron ulkopuolisista lähteistä, jolloin ulkopuolelle näyttää siltä, että nämä asiakkaat eivät tuo ladattavaa koodia. (Fielding 2000, 84–85.)

3.5 REST-rajapintojen luokittelu

Kuten jo aiemmin todettiin, kaikki verkkorajapinnat eivät ole REST-rajapintoja. On kuitenkin yleistä, että mitä tahansa verkkorajapintaa kutsutaan REST-rajapinnaksi, vaikka ne eivät täytä kaikkia REST-tyylin rajoitteita. Leonard Richardson on luonut neliportaisen asteikon, jota voidaan käyttää verkkopalveluiden luokitteluun niiden REST-tyylin rajoitteiden noudattamisen mukaan.

Richardsonin kypsyysmalliksi kutsuttu asteikko koostuu kolmesta pääteknologiasta, joiden avulla verkkopalvelun kypsyyttä arvioidaan. Kyseiset teknologiat ovat URI, HTTP ja hypermedia. Asteikon ensimmäinen tason (taso 0) täyttävät palvelut, jotka sisältävät vain yhden URI:n ja käyttävät vain yhtä HTTP-metodia, tyypillisesti POST-metodia. Tällaiset palvelut eivät täytä ainuttakaan REST-tyylin rajoitteesta.

Asteikon toisen tason (taso 1) palvelut sisältävät useita URI-polkuja, mutta käyttävät vain yhtä HTTP-metodia. Tason 2 palvelut sisältävät useita URI-polkuja ja tukevat lisäksi useita HTTP-metodeja. Tyypillisesti CRUD-rajapinnat ovat tämän tason palveluita.

Korkeimman tason (taso 3) palvelut käyttävät edellisen tason ominaisuuksien lisäksi hypermediaa sovellustilan moottorina, eli resurssien esitykset sisältävät URI-polkuja toisiin resursseihin, ja näin palvelu johdattaa käyttäjää resursseista koostuvaa polkua pitkin sovelluksen läpi. (Webber, Parastatidis & Robinson 2010, 18–20.)

Tämä Richardsonin luoma luokittelu on hyvä tapa hahmottaa REST-arkkitehtuurin elementtejä, mutta se ei ole REST-tason määritelmä itsessään. Asteikon kolmatta tasoa voidaan pitää edellytyksenä REST-tyyliä noudattavalle järjestelmälle. Tämä luokittelu on hyödyllinen työkalu REST-ajattelun perusideoiden ymmärtämiseen askel askeleelta, ja sitä tulisikin käyttää oppimisvälineenä, ei arviointityökaluna. (Fowler 18.3.2010.)

4 HTTP

HTTP (Hypertext Transfer Protocol) on sovelluskerroksen tiedonsiirtoprotokolla hajautetuille ja yhteistoiminnallisille hypermediajärjestelmille. REST-tyyli ei vaadi käyttämään HTTP-protokollaa rajapinnan tiedonsiirrossa, mutta sitä käytetään lähes aina. HTTP on avoin standardi ja IETF:n (Internet Engineering Task Force) hyväksymä, ja siksi suosittu. (Lange 2016, 21.)

Tässä luvussa käsitellään HTTP:n peruselementit: pyynnöt ja vastaukset, metodit sekä vastauskoodit.

4.1 Pyyntö ja vastaus

HTTP on tiedonsiirtoprotokolla, jonka toiminta perustuu pyyntö-vastaus-protokollaan, jossa asiakas lähettää pyynnön palvelimelle ja palvelin lähettää takaisin vastauksen. Asiakkaan lähettämä pyyntö sisältää pyynnön metodin, URI-osoitteen, HTTP-protokollaversion sekä viestin, joka sisältää tietoa asiakkaasta ja mahdollisen viestin sisällön. Palvelimen vastaus sisältää pyynnön onnistumisesta kertovan tilakoodin sekä viestin, joka sisältää tietoa palvelimesta ja mahdollisen viestin sisällön. (Fielding ym. 1999, 11.)

Liitteessä 3 on esimerkki HTTP-pyynnöstä. Ensimmäisellä rivillä on määritelty pyynnön metodi (GET), URI-osoitteen polku (/index.html) sekä käytettävä HTTP-versio (HTTP/1.1). Seuraavilla riveillä on pyynnön otsikot (Host, User-Agent, Accept jne.), jotka kertovat palvelimelle esimerkiksi missä muodossa asiakas haluaa vastauksen. Koska kyseessä on GET-pyyntö, se ei sisällä viestiä. Sen tarkoitus on vain hakea dataa palvelimelta, ja kaikki pyynnön toteuttamiseen tarvittava tieto on jo siinä. (Richardson & Ruby 2007, alaluku HTTP: Documents in Envelopes.)

Liitteessä 4 on esimerkki HTTP-vastauksesta, joka voisi olla mahdollinen vastaus liitteessä 3 esitettyyn HTTP-pyyntöön. Vastaus voidaan jakaa kolmeen osaan: vastauskoodi, vastauksen otsikot ja vastauksen sisältö. Vastauskoodi kertoo vastaanottajalle pyynnön onnistumisen tilan ja miten vastaanottajan tulee suhtautua tähän vastaukseen. Vastauskoodeja käsitellään lisää alaluvussa 4.4.

Vastauksen otsikot ovat tuomassa lisätietoa vastauksesta, kuten päivämäärän ja palvelimen. Yksi tärkeimmistä otsikoista on Content-Type, joka kertaa sisällön mediatyypin. Liitteen 4 tapauksessa sisällön mediatyyppi on text/html, eli HTML-dokumentti, jonka verkkoselain pystyy renderöimään verkkosivuksi.

Vastauksen sisältö on se, mitä pyynnöillä yleensä haetaan. Esimerkin tapauksessa vastauksen sisältö on HTML-dokumentti, joka kertoo, minkälaisen verkkosivun pyynnön lähettäjän verkkoselaimen tulee esittää. (Richardson & Ruby 2007, alaluku HTTP: Documents in Envelopes.)

4.2 Metodit

HTTP-protokolla sisältää useita erilaisia metodeja, mutta eniten käytetyt ovat GET, DELETE, POST, PUT ja HEAD (Richardson & Ruby 2007, alaluku Method Information). Tässä luvussa käsitellään niistä neljää ensimmäistä, ja samoja metodeja käytetään myös myöhemmin toteutettavassa rajapinnassa.

Mitä tahansa metodia voi käyttää mihin operaatioon tahansa, mutta REST-tyylin yhtenäisen rajapinnan rajoitteeseen kuuluvan itseään kuvaavien viestin rajoitteen mukaan HTTP-metodien muodollista merkitystä tulee noudattaa, jotta rajapinnan käyttäminen olisi mahdollisimman selkeää. (Lange 2016, 12–14.)

GET-metodin tarkoitus on hakea tietoa. GET-metodin ei tulisi tehdä mitään muuta kuin hakea URI-osoitteessa määritelty resurssi, mutta joskus sitä käytetään kuitenkin väärin, esimerkiksi lomakkeiden lähettämisen yhteydessä. Tällöin sillä voi olla huomattavia sivuvaikutuksia, kuten uuden resurssin luominen. Lomakkeiden lähettämiseen tulisi GET-metodin sijaan käyttää POST-metodia. POST-metodi on tarkoitettu uuden resurssin luomiseen URI-osoitteessa määritellyn resurssikokoelman alle (Lange 2016, 13). Taulukon 1 esimerkissä POST-metodia käytetään uuden resurssin luomiseksi customers-resurssikokoelmaan.

PUT-metodi korvaa URI-osoitteessa määritellyn resurssin uudella pyynnön sisältöosassa määritellyllä resurssin esityksellä. PUT-metodia käytetään siis koko resurssin korvaamiseen, eikä vain osittaiseen päivittämiseen. (Lange 2016, 13.) Jos halutaan päivittää vain resurssin tiettyjä osia, voidaan käyttää PATCH-metodia (Relan 2019, 8). Resurssin poistamiseen käytetään DELETE-metodia (Lange 2016, 13). Taulukossa 1 on esimerkki kaikkien edellä mainittujen metodien käytöstä.

Taulukko 1. HTTP-metodien käyttö (mukaillen Lange 2016, 14)

Tehtävä	**Metodi**	**URI-polku**
Hae asiakas	GET	/customers/{id}
Hae kaikki asiakkaat	GET	/customers
Luo uusi asiakas	POST	/customers

Tehtävä	Metodi	URI-polku
Päivitä asiakasta	PUT	/customers/{id}
Poista asiakas	DELETE	/customers/{id}

HTTP-metodit voidaan luokitella turvallisiin ja idempotentteihin, eli muuttumattomiin. Turvalliset metodit ovat sellaisia, jotka eivät muokkaa resursseja, kuten GET-metodi. Idempotentti metodi on sellainen, joka palauttaa saman tuloksen riippumatta suoritetaanko se yhden vai useamman kerran. Esimerkiksi GET- ja PUT-metodit ovat idempotentteja. (Lange 2016, 13; Relan 2019, 7.)

4.3 Vastauskoodit

HTTP-vastauskoodi on kolmenumeroinen luku, joka liitetään HTTP-vastaukseen. Koodin tarkoitus on kertoa, mitä tapahtui palvelimen käsitellessä sille lähetettyä pyyntöä. Se tuo myös lisätietoa, kuinka vastauksen sisältöön tulisi suhtauta. (Richardson ym. 2013, luku Appendix A. The Status Codex.) Erilaisia HTTP-vastauskoodeja on olemassa yli 30 (Fielding & Reschke 2014, 50–64.), joten tässä työssä käydään läpi vain osa olennaisimmista koodeista.

Vastauskoodit on jaoteltu ryhmiin niiden laadun mukaan. Koodin ryhmän kertoo sen ensimmäinen numero. Numerolla 1 alkavat koodit ovat informatiivisia, ja niitä käytetään vain HTTP-asiakkaan ja palvelimen väliseen kommunikointiin. 2:lla alkavat koodit ilmaisevat onnistuneen pyynnön. 3:lla alkavat koodit liittyvät uudelleenohjaukseen. Ne eivät tarkoita suoranaisesti pyynnön epäonnistumista, mutta asiakkaalta vaaditaan lisätoimenpiteitä pyynnön onnistumiseen. Numerolla 4 alkavat koodit kertovat epäonnistumisesta jonkin pyynnössä olevan virheen takia. Numerolla 5 alkavat vastauskoodit tarkoittavat myös pyynnön epäonnistumista, mutta vika on palvelinpuolella eikä asiakas luultavasti voi tehdä mitään asian korjaamiseksi. (Richardson ym. 2013, alaluku Families of Status Codes.)

Yleisimpiä vastauskoodeja ovat koodit 200, 201, 400, 404 ja 500. Koodi 200 kertoo onnistuneesta pyynnöstä. Esimerkiksi onnistunut GET-pyyntö vastaus sisältää koodin 200 ja viestin sisältönä on haetun resurssin esitys. Koodi 301 palautetaan, kun asiakas suorittaa tilasiirtymän, joka muuttaa resurssin URI-osoitteen toiseen. Siirron jälkeen vanhaan osoitteeseen lähetetyt pyynnöt palauttavat myös koodin 301.

Koodi 400 kertoo ongelmasta asiakaspuolella. Mahdollisen viestin sisältönä palautetaan virheviesti, joka kuvaa tarkemmin ongelmaa. Jos ongelma on sen sijaan palvelinpuolella, palautetaan koodi 500. Myös tämä vastaus voi sisältää ongelmaa kuvaavan virheviestin, mutta asiakas ei voi korjata palvelinpuolen ongelmia.

Edellä läpikäytyjen koodien lisäksi yksi hyvin yleinen koodi on 404, joka kertoo, että palvelin ei voi yhdistää pyynnössä olevaa URI-osoitetta mihinkään resurssiin. Yleensä tämä tarkoittaa, että resurssia ei ole olemassa. (Richardson ym. 2013, alaluku Appendix A. The Status Codex.)

5 Työskentelymenetelmät

Työn toteutuksena kehitettiin REST-rajapinta kuvitteellisen blogipalvelun tarpeisiin Flask-sovelluskehystä hyödyntäen. Flask on Python-ohjelmointikielelle tehty kevyt mikrosovelluskehys. Se kehitettiin alun perin vuonna 2010 avoimen lähdekoodin kehittäjätiimi Pocoon toimesta, mutta nykyisin sen kehittämisestä vastaa The Pallet Projects.

Flask koostuu kahdesta pääkomponentista, Werkzeugista ja Jinja2:sta. Wergzeug vastaa reitityksestä, virheenkorjauksesta ja WSGI:stä, kun taas Jinja2 toimii Flaskin mallimoottorina. Flask ei sisällä tietokantayhteyksiä tai käyttäjän tunnistusta, mutta se tukee laajennuksia, jotka mahdollistavat näiden toimintojen lisäämisen. Niinpä Flask on myös tuotantokäyttöön sopiva kehys verkkosovellusten kehittämiseen. Flask-sovellus voidaan mahduttaa yhteen Python-tiedostoon tai modulaarisesti useampaan tiedostoon. Idea Flaskin taustalla on olla perusta kaikenlaisille sovelluksille ja jättää kaikki muu laajennusten vastuulle. (Relan 2019, 1.)

Rajapinnan kehityksessä hyödynnettiin ketterään kehitykseen kuuluvan iteratiivisen kehityksen menetelmiä. Iteratiivisessa kehityksessä uusi ominaisuus suunnitellaan, kehitetään ja toteutetaan sekä testataan toistuvissa sykleissä. Jokaisessa syklissä on tarkoitus saada uusi ominaisuus valmiiksi osaksi kokonaisuutta. (Francino 2011.)

Rajapinnan kehitys alkoi vaatimusmäärittelyllä, jossa tarvittavat vaatimukset määriteltiin ja kirjattiin ylös (ks. liite 5). Tämän jälkeen vaatimukset jaoteltiin sykleihin, joissa pyrittiin saamaan tietty ominaisuus tai vaatimus valmiiksi. Tällainen yhden ominaisuuden toteuttaminen kerrallaan helpottaa kokonaisuuden hallitsemista, joka voi olla joskus hankalaa varsinkin yksin työskennellessä. Työn edetessä huomattiin myös joitakin lisävaatimuksia, joita ei suunnitteluvaiheessa ollut osattu miettiä. Sykleissä työskentely mahdollistaa nopean sopeutumisen muutoksiin, joten näihin muuttuneisiin vaatimuksiin pystyttiin vastaamaan ketterästi.

Rajapinnan lähdekoodin hallintaan käytettiin Git-versionhallintajärjestelmää, joka on eniten käytetty DevOps-työkalu lähdekoodin tallentamiseen ja muutosten hallintaan. Git mahdollistaa koodin tallentamisen vaiheittain sekä myös muutosten tarkastelun ja perumisen jälkeen päin. (NovelVista 27.7.2021.) Git sopii hyvin yhteen sykleissä työskentelyn kanssa, sillä siinä uuden kokonaisuuden lisääminen aiempaan lähdekoodiin on kätevää. Siksi se soveltui hyvin myös tässä työssä toteutetun rajapinnan kehitykseen, sillä ominaisuuksia lisättiin vaihe vaiheelta.

Kehitystyön tukena käytettiin Flask-kehyksen dokumentaatiota (https://flask.palletsprojects.com/en/2.2.x/), jonka ohjeita seuraamalla kehitys aloitettiin vaihe vaiheelta. Dokumentaatiosta selvisi, että Flask tarjoaa nimenomaan REST-rajapintojen

kehittämiseen tarkoitettuja lisäosia. Niitä päätettiin kuitenkin olla käyttämättä, sillä rajapinta oli mahdollista toteuttaa ilmankin, ja lukuisten lisäosien ja kirjastojen käyttöä haluttiin välttää, sillä yksi opinnäytetyön tavoitteista oli oppia Flask-kehyksen käyttöä, ja parhaiten se onnistuu ilman ylimääräisten kirjastojen käyttöä. Laajempien rajapintojen kehityksessä lisäosien ja kirjastojen käytöstä olisi kuitenkin varmasti hyötyä.

Flask, kuten myös sen lisäosat, on hyvin dokumentoitu, joten suurimpaan osaan kohdatuista haasteista löytyi vastaus dokumentaatioista. Flask on myös laajasti käytetty sovelluskehys, joten myös keskustelufoorumeilta löytyi paljon vastauksia ongelmatilanteisiin.

Rajapinnan testaukseen käytettiin Postman-ohjelmaa ja Flask-Testing-kirjastoa. Testauksessa käytettiin toiminnallisen testaamisen menetelmää. Toiminnallisen testauksen on tarkoitus varmistaa, että sovelluksen ominaisuudet käyttäytyvät kuten on tarkoitettu ja vaatimuksissa määritelty. (Zalavadia 2023.) Postman-ohjelmalla testattiin HTTP-pyyntöjen vastauksia, ja tarkistettiin käyttäjän kirjautumisen ja kirjautumista vaativien HTTP-pyyntöjen toimivuutta. Flask-Testing-kirjastolla toteutettiin automatisoituja yksikkötestejä, joissa lisättiin väliaikaiseen tietokantaan testidataa, ja varmistettiin, että se palautetaan haettaessa oikeassa muodossa.

6 REST-rajapinnan toteutus Flask-sovelluskehyksellä

Tässä luvussa käydään läpi REST-rajapinnan toteutusta Flask-sovelluskehyksellä. Työssä ei käydä läpi kehitystyötä vaihe vaiheelta, vaan keskitytään REST-tyylin kannalta olennaisiin seikkoihin rajapinnan kehityksessä. Tarkoitus on tarkastella, miten hyvin Flask pystyy vastamaan REST-tyylin rajoitteisiin ja vaatimuksiin. Toteutetun rajapinnan lähdekoodi löytyy kokonaisuudessaan osoitteesta https://github.com/petrushurtig/flask-blog-api, ja siitä saa tarkemman kuvan rajapinnan kehityksestä.

6.1 Toteutettava rajapinta

Toteutettavan rajapinnan perustoiminnallisuus on blogipostausten luominen ja niiden hakeminen. Vain kirjautunut käyttäjä voi luoda blogipostauksia, mutta postausten hakeminen ei vaadi kirjautumista. Postausta luodessaan käyttäjä voi halutessaan lisätä tunnisteita postaukseen. Näiden tunnisteiden avulla voidaan hakea kaikki postaukset, joissa on tietty tunniste. Kirjautunut käyttäjä voi myös luoda blogipostaukseen liittyviä kommentteja.

Rajapinnassa on määritelty kaksi eri käyttäjätasoa. Perustason käyttäjä voi hakea, muokata ja poistaa vain omia käyttäjätietojaan. Järjestelmänvalvoja-tason käyttäjä voi omien tietojensa lisäksi hakea, muokata ja poistaa toisten käyttäjien tietoja. Tämän toiminnallisuuden ideana on, että vain järjestelmänvalvoja voi antaa toiselle käyttäjälle järjestelmänvalvojan oikeudet. Kaikki uudet käyttäjät ovat oletuksena perustason käyttäjiä.

Käyttäjän todennukseen käytettiin JSON Web Token -standardia. Siinä HTTP-pyynnön Authorization-otsikkoon lisätään merkkijono, jolla pyynnön lähettänyt käyttäjä voidaan varmentaa. Osa pyynnöistä vaatii käyttäjältä myös tietyn käyttäjäroolin omistamista, joten sellaisten pyyntöjen kohdalla tarkastetaan ensin tietokannasta omistaako pyynnön lähettäjä vaadittavaa roolia ennen vastauksen lähettämistä.

Rajapinnan toiminnallisuudet ja muut keskeiset tiedot on lueteltu liitteessä 5. Rajapinnan toiminnallisuuksista voidaan johtaa myös rajapinnan resurssit. Resurssit ovat siis postaus, tunniste, kommentti sekä käyttäjä. Käyttäjäroolia ei luokitella tässä rajapinnassa omaksi resurssikseen, koska roolilla ei ole omia toiminnallisuuksia. Rajapinnan kaksi eri roolityyppiä määritellään ja lisätään heti tietokannan luomisvaiheessa, eikä roolityyppejä voi sen jälkeen lisätä tai muuttaa.

6.2 Flask ja REST-tyylin rajoitteet

Yksinkertaisimmillaan Flask-kehyksellä toteutettu rajapinta on liitteessä 6 kuvatun kaltainen. Tämä rajapinta palauttaa tyhjän listan, kun URI-polkuun /posts lähetään HTTP-pyyntö. Vaikka tällaiselle

rajapinnalle ei ole todellista käyttöä, sisältää se rajapinnan toteutuksen Flask-kehyksellä pienoiskoossa.

Käydään seuraavaksi läpi luvussa 3 esitellyt REST-tyylin rajoitteet, ja miten niitä noudattavan rajapinnan toteuttaminen onnistuu Flask-kehyksellä.

6.2.1 Tilaton palvelin

REST-tyylissä palvelimen tulee olla itsenäinen ja tilaton. Sen ei pidä ylläpitää tilaa asiakkaan puolesta, ja kunkin pyynnön tulee sisältää kaikki tarvittavat tiedot, jotta palvelin voi ymmärtää pyynnön ja antaa vastauksen. Jokainen pyyntö käsitellään erillisinä tapahtumina, eikä niillä ole mitään yhteyttä toisiinsa.

Liitteessä 6 esitetty pienoisohjelma on jo erillinen palvelinpuolen ohjelma, joka voi toimia itsenäisesti, ja jota voidaan jatkokehittää välittämättä asiakkaan toteutuksesta tai tilasta. Näin ollen Flask soveltuu siis rajapintojen tilattomien palvelimien kehittämiseen.

Opinnäytetyössä toteutettu rajapinta noudattaa myös tilattomuuden rajoitusta. Sen palauttamat vastaukset eivät tarvitse tietoa aiemmista kutusuista, vaan se palauttaa tiettyyn pyyntöön aina saman vastauksen. Mitään tietoa pyynnöistä ei tallenneta palvelimelle, joten sen voi sanoa olevan tilaton.

6.2.2 Yhtenäinen rajapinta

Yksi keskeinen rajoite REST-tyylissä on rajapinnan yhtenäistäminen. Se pitää sisällään resurssien tunnistamisen, resurssien käsittelyn esitysten kautta, itseään kuvaavat viestit sekä hypermedian sovellustilan moottorina.

Resurssien tunnistamisen voi Flask-kehyksellä toteuttaa määrittelemällä jokaiselle resurssille URI-polun, johon lähetetty HTTP-pyyntö koskee kyseistä resurssia. Yksi tapa toteuttaa URI-polkujen määritys on käyttää Flaskin blueprint-ominaisuutta.

Liitteessä 7 esitellyssä esimerkissä luodaan blueprint-objekti ja määritellään sille route, eli URI-polku sekä metodi tai metodit. Yhdelle blueprint-objektille voidaan määritellä useita polkuja, joten se on kätevä tapa organisoida yhden resurssin kaikki polut yhteen objektiin. Blueprint-objekti tulee myös rekisteröidä Flask sovellukseen, ja samalla sille annetaan URI-polun prefiksi. (Liite 7.)

Liitteessä 7 kuvatulla tavalla muodostettu URI-polku on siis ”/posts/”, johon lähetetty GET-pyyntö palauttaa blogipostausten kokoelmaresurssin tietokannasta. Yksittäisen postauksen resurssin URI

voidaan määritellä lisäämällä blueprint-objektin route:ksi ”/<post_id>”. Näin ollen resurssien tunnistaminen URI-osoitteiden avulla onnistuu Flask-kehyksellä hyvin.

Myös esimerkki resurssien käsittelystä esitysten kautta voidaan nähdä get_all_posts-metodissa (liite 7). Siinä postaukset palautetaan Flaskin jsonify-funktion avulla, joka muuttaa palautettavan blogipostausten datan JSON-muotoon. Asiakkaan lähettämä GET-pyyntö käynnistää ketjun, jossa pyyntö välitetään rajapinnan eri komponenttien välillä aina tietokantaluokalle asti, joka suorittaa varsinaisen tietokantakyselyn. Tämän tietokantakyselyn tuloksena saatu resurssi palautetaan asiakkaalle JSON-muotoisen esityksenä (liite 8).

6.2.3 Itseään kuvaavat viestit

REST-tyylin mukaan HTTP-metodien muodollista merkitystä tulee noudattaa, jotta rajapinnan käyttö olisi mahdollisimman selkeää. Toteutetussa rajapinnassa tätä rajoitetta on noudatettu hyvin pitkälle. GET-metodilla haetaan, POST-metodilla luodaan, PUT-metodilla korvataan ja päivitetään ja DELETE-metodilla poistetaan resursseja. Yksi poikkeus on kuitenkin /posts/<post_id> -resurssi, jolla haetaan yksittäinen blogipostaus, sillä siinä resurssin views-saraketta kasvatetaan yhdellä joka kerta kun metodia kutsutaan. Tämän tarkoitus on mahdollistaa blogipostausten lajittelu eniten luettujen mukaan. (Liite 9.)

Samassa koodissa voidaan nähdä esimerkki HTTP-vastauskoodien käytöstä Flask-kehyksessä. Jos URI-osoitteessa määritellyllä tunnisteella ei löydy postausta, palautetaan koodi 404. Jos taas resurssi löytyy, palautetaan onnistuneesta pyynnöstä kertova koodi 200. Jos tapahtuu joku muu virhe, palautetaan palvelinpuolen virheestä kertova koodi 500. (Liite 9.)

Yksi REST-tyyliä noudattavan rajapinnan ominaisuuksista on hyperteksti sovellustilan moottorina, joka tarkoittaa esimerkiksi resurssiin liittyvien linkkien liittämistä palvelimen lähettämiin vastauksiin. Flask-kehyksessä palautettavaa vastausta voi muokata hyvin vapaasti, joten siihen voi lisätä myös hypertekstiä. Yksi tapa tähän on lisätä linkit tietokantaluokan metodiin, joka palauttaa resurssin JSON-muodossa.

Liitteessä 10 kuvatussa tietokantaluokan metodissa yksi blogipostaus palautetaan JSON-muodossa. Jos metodin argumentti ”links” saa arvon True, liitetään metodin palautukseen linkit postauksen kommentteihin ja tunnisteisiin. Jos taas links-argumentin arvo on False, liitetään postauksen mahdolliset kommentit ja tunnisteet palautukseen objektilistana.

6.2.4 Välimuisti

REST-tyylin rajoitusten mukaan palvelimen vastausten tulee sisältää tieto vastauksen välimuistituksesta. Flask-sovelluksessa tiedon välimuistituksesta voi asettaa joko kaikkien vastausten HTTP-otsakkeisiin, tai jokaiselle vastaukselle erikseen.

Liitteessä 11 on esimerkki kaikkiin vastauksiin lisätystä HTTP-otsakkeesta, jossa välimuistitus sallitaan tietyksi ajaksi. HTTP-otsake Cache-Control on ohje, joka määrittelee kuinka kauan vastauksen pitäisi olla välimuistissa. Esimerkin tapauksessa asiakas voi siis tallettaa vastauksen 3600 sekunniksi, ja käyttää samaa vastausta uudelleen, jos samanlainen pyyntö lähetetään 3600 sekunnin kuluessa. Tämä vähentää palvelimelle kohdistuvaa kuormitusta ja nopeuttaa myös asiakkaan toimintaa.

6.2.5 Kerroksellinen järjestelmä ja ladattava koodi

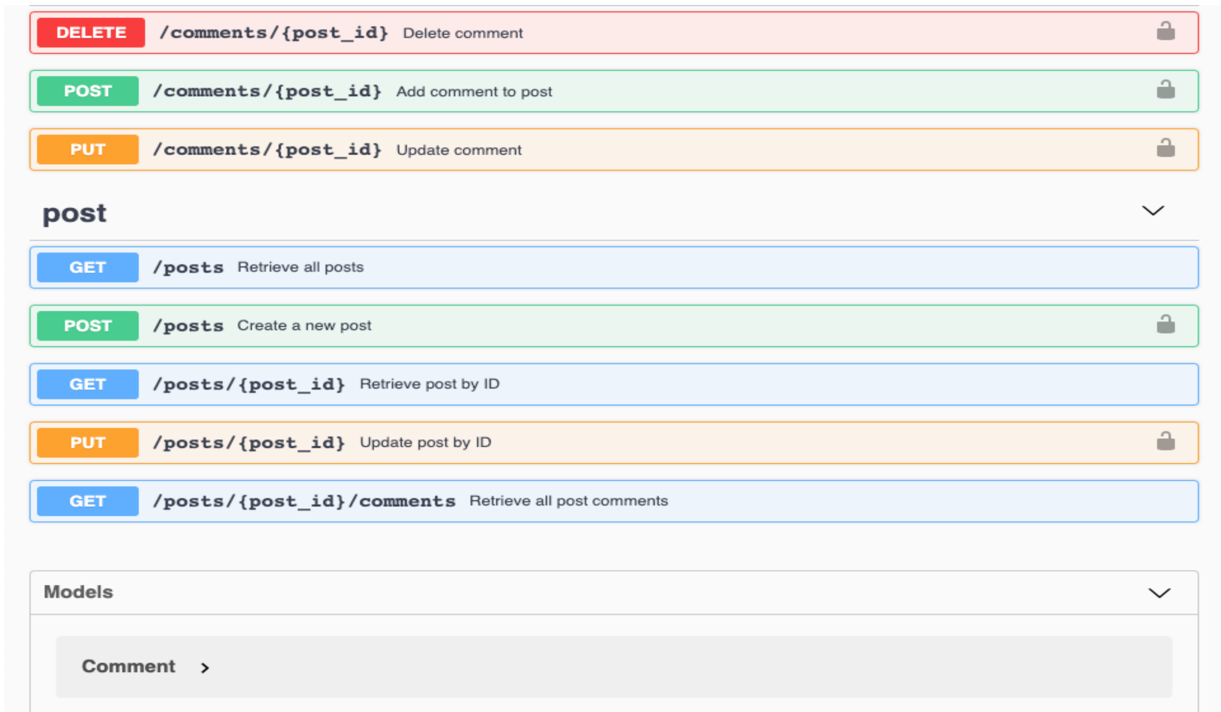
REST-arkkitehtuurityylissä järjestelmät tulee rakentaa kerroksittain siten, että yksi kerros on vuorovaikutuksessa ainoastaan naapurikerrostensa kanssa. Toteutetussa rajapinnassa tämä käytännössä toteutuu, sillä kerroksia on vain kaksi.

REST-tyylin valinnaisen ladattava koodi -rajoituksen mukaan asiakas voi ladata palvelimelta koodia laajentamaan asiakaspuolen toiminnallisuutta. Toteutuksen rajapinnassa ei nähty käyttöä tällaiselle palvelimelta ladattavalle koodille. Flask tarjoaa kuitenkin mahdollisuuden tällaiselle ladattavalle koodille, koska sen palauttamat vastaukset ovat hyvin vapaasti muokattavissa.

6.2.6 Dokumentaatio

Toteutetun rajapinnan dokumentaatio toteutettiin Flasgger-kirjaston avulla, joka helpottaa Swagger-dokumentaation luomista Flask-sovelluksessa. Liitteessä 12 on nähtävissä, miten Swagger alustetaan Flask-sovellukseen ja määritellään tiedosto, josta Swagger saa dokumentaation tiedot.

Varsinaisen dokumentaation kirjoittamiseen voi käyttää montaa tapaa. Toteutuksessa käytettiin YAML-tiedostoa, jossa kuvataan jokainen rajapinnan resurssi ja niiden URI-osoitteet YAML-muodossa. Liitteessä 13 on osa blogipostaus-resurssin ja yhden sen URI-osoitteen määrityksistä YAML-muodossa. Valmiissa Swagger-dokumentaatiossa on lueteltu rajapinnan resurssit ja niiden URI-osoitteet, ja pyyntöjen lähettämistä niihin pystyy myös kokeilemaan käytännössä (kuva 2).



Kuva 2. Ruutukaappaus Swagger-dokumentaatiosta

7 Yhteenveto

Tässä työssä tutustuttiin ohjelmointirajapintoihin ja erityisesti REST-arkkitehtuurityyliin verkkorajapintojen kehityksessä. Työssä käytiin läpi REST-tyylin taustaa ja käyttöä sekä sen asettamia rajoituksia rajapinnoille. Työssä sivuttiin myös HTTP-protokollan toimintaa ja käsitteistöä.

Työn empiirisessä osassa toteutettiin REST-tyyliä noudattava verkkorajapinta kuvitteelliselle blogipalvelulle Flask-kehystä hyödyntäen. Toteutusvaiheen tarkoitus oli tutkia, miten REST-rajapinta toteutetaan käytännössä, ja miten Flask-kehys sopii sellaisen toteuttamiseen.

Työn ensimmäinen tavoite oli selvittää, mitä REST-tyyliä noudattava rajapinta tarkoittaa ja mitä hyötyä on REST-tyylin noudattamisesta. Näihin kysymyksiin saatiin vastaus työn teoriaosuuden luvussa 3.

Toinen tavoite oli tutkia, miten Flask-kehys soveltuu REST-tyyliä noudattavan rajapinnan toteuttamiseen, ja työssä havaittiin, että Flask sopii hyvin yksinkertaisen REST-tyyliä noudattavan verkkorajapinnan toteuttamiseen. Vaikka se ei ole erityisesti REST-tyyliä noudattavien rajapintojen kehitykseen tarkoitettu kehys, voi sillä toteuttaa kaikkiin REST-tyylin rajoitteisiin vastaavan rajapinnan.

Toteutettu rajapinta on REST-tyylin asiakas-palvelin-mallin mukainen yhtenäinen rajapinta. Myös tilattomuus sekä järjestelmän kerroksellisuus on huomioitu. REST-tyylin valinnaista ladattavan koodin rajoitetta ei rajapinnassa toteutettu, koska sille ei nähty tarvetta tässä rajapinnassa. Sen lisääminen olisi kuitenkin mahdollista Flask-kehyksen dokumentaation mukaan.

Näin yksinkertaisessa rajapinnassa REST-tyylin mukaisten käytänteiden noudattaminen Flask-kehyksellä on melko helppoa, mutta rajapinnan laajentuessa se voi muuttua hankalammaksi ja työläämmäksi. Silloin erilaisten REST-rajapintoihin erikoistuneiden Flask-kehyksen lisäosien ja muiden kirjastojen käyttö on suositeltavaa.

Toteutettu rajapinta ei ole täydellinen rajapinta, eikä se täytä kokonaan kaikkia REST-tyylin rajoitteita. Esimerkiksi yhtenäisen rajapinnan rajoitteeseen kuuluva vaatimus hypermediasta sovellustilan moottorina ei toteudu täysimääräisesti toteutetussa rajapinnassa. Toteutuksen tavoitteena oli kuitenkin vain selvittää, onnistuisiko REST-tyyliä noudattavan rajapinnan toteutus Flask-kehyksellä, ja siihen tässä työssä saatiin vastaus. Rajapintaa tulisi edelleen jatkokehittää ja testata oikealla asiakaspuolen toteutuksella, sillä näin huomattaisiin rajapinnan mahdolliset puutteet ja niitä voitaisiin parannella ja korjata.

Työn kolmas tavoite oli oppia lisää REST-tyylistä ja sovelluskehityksestä Flask-kehyksellä. Myös tämä tavoite toteutui, sillä nyt tiedän, mitkä ovat REST-tyylin rajoitteet ja hyödyt, ja minulla on nyt ensimmäiset kokemukset ohjelmistokehityksestä Flask-kehyksellä.

Lähteet

Bigelow, S. 8.2.2022. What are the types of APIs and their differences? TechTarget. Verkkosivu. Luettavissa: https://www.techtarget.com/searchapparchitecture/tip/What-are-the-types-of-APIs-and-their-differences. Luettu: 19.12.2022.

De, B. 2017. API Management: An Architect's Guide to Developing And Managing APIs for Your Organization. Apress. New York. E-kirja. Luettu: 14.12.2022.

Fowler, M. 18.3.2010. Richardson Maturity Model. Martin Fowlerin blogi. Luettavissa: https://martinfowler.com/articles/richardsonMaturityModel.html. Luettu: 10.4.2023.

Fielding, R. 2000. Architectural Styles and the Design of Network-based Software Architectures. Väitöskirja. Kalifornian yliopisto. Luettavissa: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. Luettu: 1.12.2022.

Fielding, R. 2008. REST APIs must be hypertext-driven. Roy Fieldingin blogi. Luettavissa: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. Luettu: 17.12.2022.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masiner, L., Leach, P. & Berners-Lee, T. 1999. Hypertext Transfer Protocol -- HTTP/1.1. Internet Engineering Task Force. Luettu: 11.12.2022. Luettavissa: https://www.rfc-editor.org/rfc/rfc2616.txt.

Fielding, R. & Reschke, J. 2014. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. Internet Engineering Task Force. Luettu: 4.2.2023. Luettavissa: https://www.rfc-editor.org/rfc/rfc7231.

Francino, Y. 2011. Definition: iterative development. TechTarget. Luettavissa: https://www.techtarget.com/searchsoftwarequality/definition/iterative-development. Luettu: 14.3.2023.

Gupta, L. 7.4.2022. What is REST. Restfulapi blogi. Luettavissa: https://restfulapi.net/. Luettu: 8.4.2023.

Jin, B., Sahni, S. & Shevat, A. 2018. Designing Web APIs. O'Reilly Media, Inc. Sebastopol, Kalifornia. E-kirja. Luettu: 19.12.2022.

Kaushik, G. 2022. Rest Architectural Elements. ToolsQA. Luettavissa: https://www.toolsqa.com/rest-assured/rest-architectural-elements/. Luettu: 8.4.2023.

Lane, K. 26.9.2019. Intro to APIs: What Are APIs Used for? Postman blogi. Luettavissa: https://blog.postman.com/intro-to-apis-what-are-apis-used-for. Luettu: 19.12.2022.

Lane, K. 10.10.2019. Intro to APIs: History of APIs. Postman blogi. Luettavissa: https://blog.postman.com/intro-to-apis-history-of-apis/. Luettu 19.12.2022.

Lange, K. 2016. The little book on REST services. Kööpenhamina. Luettavissa: https://www.kennethlange.com/books/The-Little-Book-on-REST-Services.pdf. Luettu: 11.12.2022.

Nadkarni, S. 2022. What is REST? ToolsQA. Luettavissa: https://www.toolsqa.com/rest-assured/what-is-rest/. Luettu: 8.4.2023.

NovelVista 27.7.2021. Git! An Important DevOps Tool. NovelVista blogi. Luettavissa: https://www.novelvista.com/blogs/devops/git-an-important-devops-tool. Luettu: 8.3.2023.

Pedro, B. 2017. What are Web APIs. Luettavissa: https://hackernoon.com/what-are-web-apis-c74053fa4072. Luettu: 10.12.2022

Postman 2022. 2022 State of the API Report. Luettavissa: https://www.postman.com/state-of-api/api-technologies. Luettu: 19.12.2022.

Red Hat, 2019. REST vs. SOAP. Luettavissa: https://www.redhat.com/en/topics/integration/whats-the-difference-between-soap-rest. Luettu: 19.12.2022.

Red Hat, 2022. What is an API? Luettavissa: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces. Luettu: 10.12.2022.

Relan, K. 2019. Building REST APIs with Flask. Apress. New York. E-kirja. Luettu: 1.12.2022.

Richardson, L. & Ruby, S. 2007. RESTful Web Services. O'Reilly Media, Inc. Sebastopol, Kalifornia. E-kirja. Luettu: 11.12.2022.

Richardson, L., Amundsen, M. & Ruby, S. 2013. RESTful Web APIs. O'Reilly Media, Inc. Sebastopol, Kalifornia. E-kirja. Luettu: 17.12.2022.

Simpson, J. 15.3.2022. 6 Types of APIs: Open, Public, Partner, Private, Composite, Unified. Nordic APIS blogi. Luettavissa: https://nordicapis.com/6-types-of-apis-open-public-partner-private-composite-unified/. Luettu: 3.4.2023.

Stowe, M. 2014. API Best Practices: Hypermedia (Part 4.1). Luettavissa: https://blogs.mulesoft.com/dev-guides/api-design/api-best-practices-hypermedia-part-1/. Luettu 17.12.2022.

Wagner, J. Understanding the Differences Between API Documentation, Specifications, and Definitions. Swagger. Luettavissa: https://swagger.io/resources/articles/difference-between-api-documentation-specification/. Luettu: 22.12.2022.

Webber, J., Parastatidis, S. & Robinson, I. 2010. REST in Practice. O'Reilly Media, Inc. Sebastopol, Kalifornia. Luettu: 8.4.2023.

Wilfred, T. 3.9.2022. Different Types Of API For Web Development 2021. Codesera blogi. Luettavissa: https://codersera.com/blog/different-types-of-api-for-web-development/. Luettu: 19.12.2022.

Zalavadia, S. 2023. Complete Functional Testing Guide With Its Types And Example. Software Testing Help. Luettavissa: https://www.softwaretestinghelp.com/guide-to-functional-testing/. Luettu: 3.4.2023.

Liitteet

Liite 1. Resurssin esitys JSON-muodossa

```
{
    "title": "Näitä neljää asiaa et tiennyt Pythonista",
    "content": ”1.Python on alun perin suunniteltu matematiikkaan ja tutkimukseen 2. Pythonilla on yli 140 000 avoimen lähdekoodin kirjastoa ja pakettia 3. Pythonilla on erittäin laaja käyttöalue erilaisissa sovellusalueissa, kuten tiedon hallinnassa, pelikehityksessä, tekoälyssä ja robotiikassa 4. Python on yksi maailman suosituimmista ohjelmointikielistä ja sillä on aktiivinen ja laaja kehittäjäyhteisö.”
}
```

Liite 2. Hypertekstiä sisältävä resurssin esitys


```
{
  "id": 12,
  "firstname": "Han",
  "lastname": "Solo",
  "_links": {
    "self": {
      "href": "https://api.example.com/customers/12"
    },
    "orders": {
      "href": "https://api.example.com/orders?customerId=12"
    }
  }
}
```

Liite 3. HTTP-pyyntö (Richardson & Ruby 2007, alaluku HTTP: Documents in Envelopes)

```
GET /index.html HTTP/1.1

Host: www.oreilly.com

User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.12)...

Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,...

Accept-Language: us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-15,utf-8;q=0.7,*;q=0.7

Keep-Alive: 300

Connection: keep-alive
```

Liite 4. HTTP-vastaus (Richardson & Ruby 2007, alaluku HTTP: Documents in Envelopes)

```
HTTP/1.1 200 OK

Date: Fri, 17 Nov 2006 15:36:32 GMT

Server: Apache

Last-Modified: Fri, 17 Nov 2006 09:05:32 GMT

Etag: "7359b7-a7fa-455d8264

Accept-Ranges: bytes

Content-Length: 43302

Content-Type: text/html

X-Cache: MISS from www.oreilly.com

Keep-Alive: timeout=15, max=1000

Connection: Keep-Alive


<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>

...

<title>oreilly.com -- Welcome to O'Reilly Media, Inc.</title>
...
```

Liite 5. Rajapinnan toiminnot lisätietoineen

Toiminto	URI-polku	Parametrit	HTTP-metodi	Vaatii autentikaation
Hae kaikki postaukset	/posts	Ei	GET	Ei
Hae yksi postaus	/posts/<post_id>	Blogipostauksen id	GET	Ei
Luo uusi postaus	/posts	Ei	POST	Kyllä
Muokkaa postausta	/posts/<post_id>	Blogipostauksen id	PUT	Kyllä
Poista postaus	/posts/<post_id>	Blogipostauksen id	DELETE	Kyllä
Hae yksi käyttäjä	/users/<user_id>	Käyttäjän id	GET	Kyllä
Luo uusi käyttäjä	/users	Ei	POST	Ei
Muokkaa käyttäjää	/users/<user_id>	Käyttäjän id	PUT	Kyllä
Poista käyttäjä	/users/<user_id>	Käyttäjän id	DELETE	Kyllä
Lisää kommentti blogipostaukseen	/posts/<post_id>/comments	Blogipostauksen id	POST	Kyllä
Muokkaa kommenttia	/comments/<comment_id>	Kommentin id	PUT	Kyllä
Poista kommentti	/comments/<comment_id>	Kommentin id	DELETE	Kyllä

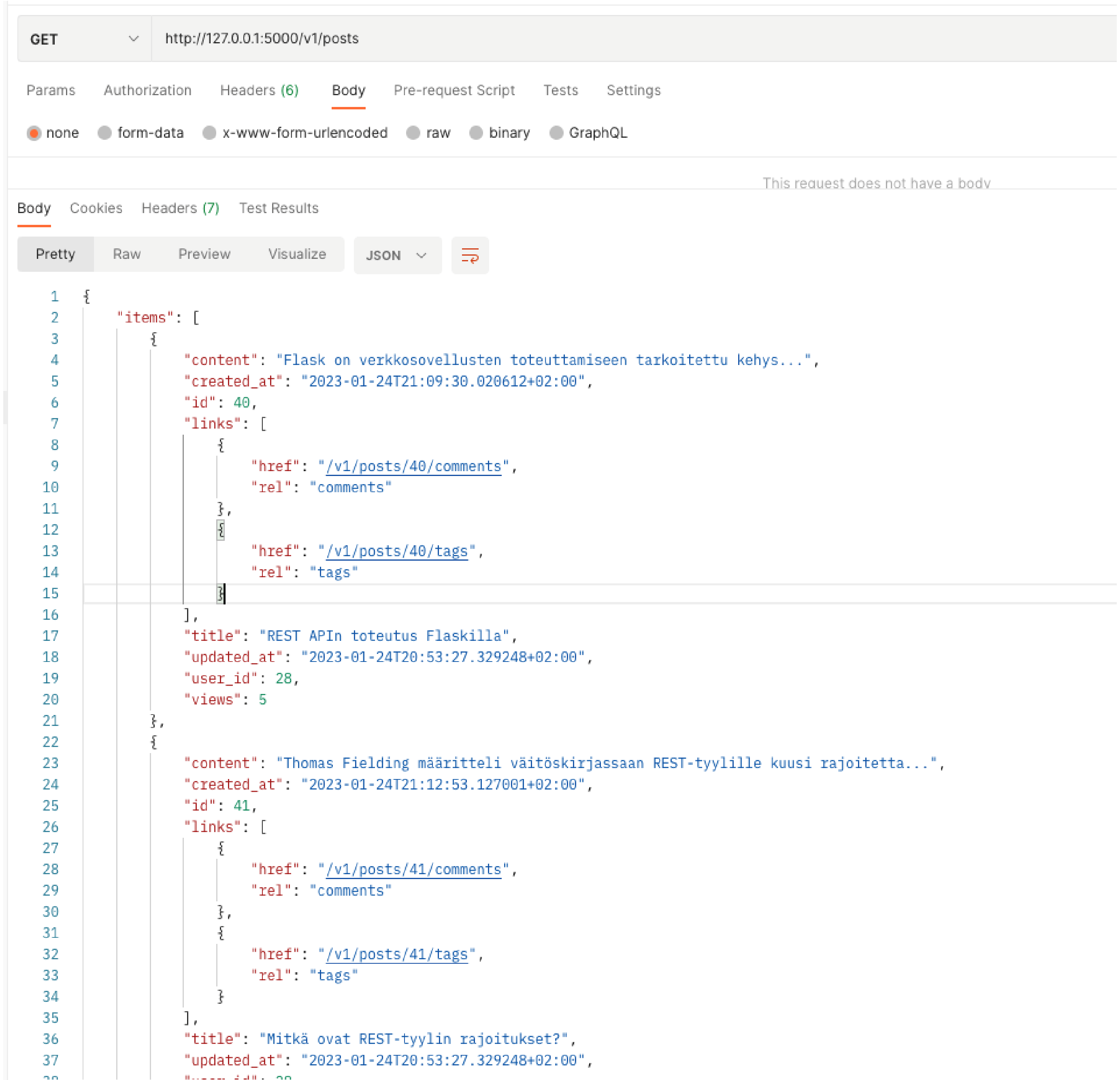
Liite 6. Flask-kehyksellä tehty minimalistinen rajapinta

```
from flask import Flask

app = Flask(__name__)

@app.route("/posts")

def get_posts():

    return []
```

Liite 7. Flaskin blueprint-objektin routet, eli päätepisteet

```
blueprint = Blueprint("post_api", __name__)
@blueprint.route("/", methods=["GET"])
@inject
def get_all_posts(
    post_service: PostService = Provide[Container.post_service]
):
    try:
        page = request.args.get("page", 1, type=int)
        per_page = request.args.get("per_page", 10, type=int)

        posts = post_service.get_all_posts(page, per_page)

        return jsonify(pagination_links(posts, page, per_page, '.get_all_posts')), 200
    except Exception as e:
        app.logger.info(e)

        return jsonify({"message": "Server error"}), 500



#blueprintin rekisteröinti Flask-sovellukseen
app.register_blueprint(post_routes.blueprint, url_prefix='/v1/posts')
```

Liite 8. GET-pyynnön vastaus JSON-muodossa



Liite 9. Yhden blogipostaus-resurssin palauttavan päätepisteen toteutus

```
@blueprint.route("/<post_id>", methods=["GET"])
@inject
def get_post_by_id(
    post_id: int,
    post_service: PostService = Provide[Container.post_service]
):
    try:
        post: IPost = post_service.get_post_by_id(post_id)
        if not post:
            msg = {"message": "Post not found"}
            return jsonify(msg), 404

        #add +1 to post.views every time it is fetched
        post_service.increment_views(post_id)

        return jsonify(post.json()), 200

    except Exception as e:
        app.logger.info(e)
        msg = {"message": "Server error"}
        return jsonify(msg), 500
```

Liite 10. Blogipostauksen tietokantaluokan resurssin esityksen palauttava osa

```
def json(self, links: bool = False) -> dict:
    created_at = self.created_at
    updated_at = self.updated_at

    if created_at is not None:
        created_at = created_at.isoformat()
    if updated_at is not None:
        updated_at = updated_at.isoformat()

    json_dict = {
        "id": self.id,
        "user_id": self.user_id,
        "title": self.title,
        "content": self.content,
        "views": self.views,
        "created_at": created_at,
        "updated_at": updated_at
    }

    if links:
        json_dict["links"] = [
        {"rel": "comments", "href": f"/v1/posts/{self.id}/comments"},
        {"rel": "tags", "href": f"/v1/posts/{self.id}/tags"}]

    if self.comments and len(self.comments) and not links:
        json_dict["comments"] = []

        for comment in self.comments:
            json_dict["comments"].append(comment.json())
```

```
if self.tags and len(self.tags) and not links:
    json_dict["tags"] = []

    for tag in self.tags:
        json_dict["tags"].append(tag.json())

return json_dict
```

Liite 11. Välimuistituksen sallivan tiedon lisääminen HTTP-vastauksiin

```
from flask import Flask, Response

app = Flask(__name__)

response = Response()

@app.after_request
def add_header(response):
    response.cache_control.max_age = 3600
    return response
```

Liite 12. Swagger-dokumentaation alustaminen

```
from flasgger import Swagger

from flask import Flask, redirect, Response


app = Flask(__name__)


swagger = Swagger(app, template_file="api.yml")
```

Liite 13. Osa Swagger-dokumentaatiosta YAML-muodossa


```yaml
definitions:
  Post:
    type: object
    properties:
      id:
        type: integer
      title:
        type: string
      content:
        type: string
      user_id:
        type: integer
      views:
        type: integer
      created_at:
        type: string
        format: date-time
      updated_at:
        type: string
        format: date-time
      comments:
        type: array
        items:
          $ref: '#/definitions/Comment'
      tags:
        type: array
        items:
          $ref: '#/definitions/Tag'
    required:
      - title
```

```
        - content
        - user_id
paths:
  /posts:
    get:
      tags:
        - post
      summary: Retrieve all posts
      parameters:
        - name: page
          in: query
          schema:
            type: integer
            default: 1
        - name: per_page
          in: query
          schema:
            type: integer
            default: 10
      responses:
        200:
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/definitions/Post'
```