Tampereen ammattikorkeakoulu

OpenAPI-spesifikaation mukaisten rajapintojen toteuttaminen Djangolla ja Django REST frameworkilla

Teemu Viikeri

OPINNÄYTETYÖ
Kesäkuu 2025

Tietojenkäsittelyn tradenomi
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn tradenomi
Ohjelmistotuotanto

VIIKERI, TEEMU:
OpenAPI-spesifikaation mukaisten rajapintojen toteuttaminen Djangolla ja Django REST frameworkilla

Opinnäytetyö 75 sivua, joista liitteitä 2 sivua
Kesäkuu 2025

Tämän opinnäytetyön tavoitteena oli tutkia ja selventää OpenAPI-spesifikaation mukaisten rajapintojen toteutusta, kuvaamista ja dokumentointia Django-verkkosovelluskehyksen ja erityisesti Django REST framework -kehyksen avulla. Työssä perehdyttiin rajapintojen yleiseen merkitykseen ohjelmistokehityksessä, HTTP-protokollan toimintaperiaatteisiin ja REST-arkkitehtuurityyliin.

Opinnäytetyössä käsiteltiin OpenAPI-spesifikaatioon liittyvää historiaa, rakennetta ja sen tarjoamia hyötyjä rajapintojen suunnittelussa ja hallinnassa. Erityistä huomiota kiinnitettiin OpenAPI-spesifikaation ekosysteemin työkaluihin, kuten Swagger UI:n automaattiseen dokumentaation generointiin, Prismin kaltaisiin validaatio- ja jäljittelypalvelimien luontityökaluihin sekä OpenAPI Generatorin kykyyn luoda SDK-kirjastoja ja palvelinrunkoja, jotka nopeuttavat kehitysprosessia.

Työssä selvitettiin myös Djangon ja Django REST frameworkin toimintaa ja roolia web-sovellusten ja REST-rajapintojen rakentamisessa. Django Ninja esiteltiin yhtenä Django REST frameworkin vaihtoehtona sen hyötyjen vuoksi.

Opinnäytetyössä pyrittiin tuottamaan konkreettista lisäarvoa työn tilaajalle Haltu Oy:lle, jonka tarpeena oli saada rajapintasovelluskehityksen yhteyteen parempaa rajapintadokumentaatiota. Tätä lähdettiin toteuttamaan kehittämällä keskitetysti käytössä olevaa lähdekoodin mallipohjaa, jota kuka tahansa yrityksessä pystyy hyödyntämään. Kehitetty mallipohja sisältää kaikki tarvittavat osat automaattisen OpenAPI-dokumentaation luomiseen osana Django REST frameworkia hyödyntävää Django-sovellusta. Käytännön sovellus kuvaa etenkin mallipohjan kehittämisen prosessia ja toteutukseen vaikuttavia seikkoja.

Pohdinta-osiossa käsitellään työn johtopäätöksiä ja mahdollisia jatkokehitys ja -tutkimusmahdollisuuksia, jotka liittyvät OpenAPI-spesifikaation hyödyntämiseen Django- ja Django REST framework -projekteissa.

Asiasanat: rajapinta, api, openapi, django, drf

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Software Production

VIIKERI, TEEMU:
Implementing OpenAPI Specification Compliant APIs with Django and Django REST framework

Bachelor's thesis 75 pages, appendices 2 pages
June 2025

This thesis aimed to investigate and clarify implementing OpenAPI Specification compliant APIs using Django and Django REST framework. The study explored APIs, the HTTP protocol, and the REST architectural style.

The thesis covers the history, structure, and benefits of the OpenAPI Specification in API design and management. It also highlights some OpenAPI Specification ecosystem tools: Swagger UI for automatic documentation, Prism for validation and mock servers, and OpenAPI Generator for creating SDKs and server stubs to speed up development.

The thesis examines how Django and Django REST framework function when building web applications and REST APIs. Django Ninja is introduced as an alternative to Django REST framework due to its advantages.

To address Haltu Oy’s needs for better API documentation, a reusable source code template was developed for anyone to use in the company. The template includes all necessary components for creating automatic OpenAPI documentation within a Django application using the Django REST framework. The thesis discusses the template’s development process and influencing factors.

Key words: api, openapi, django, drf

SISÄLLYS

LYHENTEET JA TERMIT

API	Application programming interface; sovelluksen rajapinta, minkä avulla sovellusta voidaan käyttää
AWS	Amazon Web Services; Amazonin pilvipalvelualusta
HTML	Hypertext Markup Language; merkintäkieli, jolla internetin verkkosivut on kirjoitettu. Sisältää hypertekstiä eli hyperlinkkejä sisältävää tekstiä ja tekstin rakenteen.
IETF	Internet Engineering Task Force; Internetin protokollien standardoinnista vastaava organisaatio.
JSON	JavaScript Object Notation; tiedonvälitykseen ja tallentamiseen tarkoitettu tiedostomuoto, jonka sisältö perustuu JavaScriptin objektisyntaksiin
JWT	JSON Web Token; käyttäjän kirjautumisen yhteydessä käytettävä menetelmä sekä käyttäjän tietojen välttämiseen että käyttäjän oikeuksien tunnistamiseen
MIME	Multipurpose Internet Mail Extensions; tyyppi, jolla kuvataan, millaista sisältöä viesti sisältää. Käytetään etenkin HTTP-viestinnässä.
OAS	OpenAPI Specification; spesifikaatio, joka kuvaa rakenteen, millä rajapintoja voidaan kuvata rajapinnan dokumentointia ja työkaluja varten
REST	Representational state transfer; standardoitu arkkitehtuurityyli rajapintojen suunnitteluun ja luomiseen, joka hyödyntää HTTP-protokollaa
SPA	Single-page application; verkkosovellus, joka lataa vain yhden HTML-sivun ja päivittää sen sisältöä siten, että koko sivua ei tarvitse ladata uudelleen
URI	Uniform Resource Identifier; yleinen tunniste, jolla yksilöidään resurssi verkon yli
URL	Uniform Resource Locator; resurssin osoite internetissä

XML	Extensible Markup Language; tiedonvälitykseen ja tallentamiseen tarkoitettu tiedostomuoto, joka käyttää rakenteellisia elementtejä tiedon määrittelyyn
WADL	Web Application Description Language; XML-pohjainen kieli REST-pohjaisten rajapintojen kuvaamiseen
WSDL	Web Service Description Language: XML-pohjainen kieli SOAP-pohjaisten verkkopalvelujen kuvaamiseen

1. JOHDANTO

Tietojärjestelmien välinen kommunikaatio ja integrointi ovat nykypäivän ohjelmistokehityksen keskeisiä teemoja ja haasteita. Modernissa ohjelmistomaailmassa on yhä tärkeämpää, että eri sovellukset ja palvelut voivat tehokkaasti kommunikoida keskenään. Web-pohjaiset rajapinnat ovat yksi keskeinen tekniikka tällaisen kommunikaation mahdollistamiseksi. Rajapintojen suunnittelussa ja toteutuksessa yhdeksi standardiksi on noussut OpenAPI-spesifikaatio, joka määrittelee, kuinka rajapintaa kuvaava dokumentaatio tulisi rakentaa.

Django on korkean tason websovelluskehys, joka mahdollistaa nopean websovellusten kehityksen. Django REST framework on puolestaan kirjasto, joka on erityisesti suunniteltu rakentamaan REST-pohjaisia rajapintoja Django-sovelluksissa.

Tämän opinnäytetyön tavoitteena on tutkia, miten OpenAPI-spesifikaation mukaisten rajapintojen implementointi voidaan toteuttaa Djangon ja Django REST frameworkin avulla. Työssä perehdytään siihen, miten nämä teknologiat tukevat rajapinnan kuvaamista ja dokumentointia OpenAPI-standardin mukaisesti. Keskeisessä osassa on myös esimerkkejä OpenAPI-spesifikaation ympärille rakennetun ekosysteemin työkaluista, jotka nopeuttavat ja hyödyttävät kehittäjiä rajapintojen kehityksessä.

Tältä pohjalta opinnäytetyö kuvaa käytännön sovelluksen siitä, miten työn tilaajalle luodaan keskitetty lähdekoodin mallitemplaatti OpenAPI-spesifikaation implementoinnista osana Django-sovellusta rajapintadokumentoinnin automaattisen luomiseksi.

Opinnäytetyö kuvaa vielä lyhyesti jatkokehitys- ja tutkimusmahdollisuuksia käytännön sovelluksen ja opinnäytetyön teoreettisen osuuden jatkeeksi.

2. TEOREETTINEN VIITEKEHYS

Tässä osiossa käsitellään tämän opinnäytetyön teoreettista taustaa. Tavoitteena on, että teoriaosuus kuljettaa lukijan yleisestä rajapintateoriasta kohti soveltavaa käytäntöä ja implementointia ohjelmointiesimerkkien avulla. Teoriaosuus alkaa siitä, mitä rajapinnat ovat, mikä on REST, mihin tarpeeseen ne vastaavat ja mikä rajapintojen suhde on muuhun sovelluskehitykseen. Tämän jälkeen käydään läpi OpenAPI-spesifikaatiota ja mitä hyötyjä OpenAPI-standardisoitu rajapinta tuo rajapinnan kehittäjälle sekä sen kuluttajalle. Tässä kappaleessa tutustutaan myös OpenAPI-spesifikaation ympärille rakentuneeseen ekosysteemiin. Loput kappaleet keskittyvät siihen, miten Djangolla ja Django Ninjalla rakennetut rajapinnat voidaan rakentaa OpenAPI-spesifikaation mukaisesti.

2.1. Rajapinnat

Tietotekniikan termitalkoiden (2014) mukaan rajapinta on “standardin mukainen käytäntö tai yhtymäkohta, joka mahdollistaa tietojen siirron laitteiden, ohjelmien tai käyttäjän välillä.” Tämä onkin syystä lavea määritelmä, sillä sovelluskehityksen yhteydessä käytettyjä rajapintoja löytyy hyvin erityyppisiin ja erikokoisiin tarkoituksiin. Kirjastorajapinnat esimerkiksi kertovat, miten ohjelmointikieliä tai ohjelmointikielillä rakennettuja kirjastoja käytetään muun muassa funktioiden ja luokkien kautta (Yhdysvaltain korkein oikeus 2018, 3–4). Käyttöjärjestelmärajapinnat kertovat, miten käyttöjärjestelmän ohjelmat voivat keskustella käyttöjärjestelmän kanssa ja käyttää järjestelmän resursseja (Yhdysvaltain piirituomioistuin Columbiassa 2000). Verkkorajapinnat puolestaan tarjoavat verkkopalvelun resursseja ja palveluita käyttäjille verkon ja selaimen yli (Amazon Web Services n.d.). Tämä opinnäytetyö keskittyy näistä esimerkeistä viimeisimpään eli verkkorajapintoihin. Jatkossa, kun tässä työssä puhutaan rajapinnoista, sillä tarkoitetaan verkkorajapintoja.

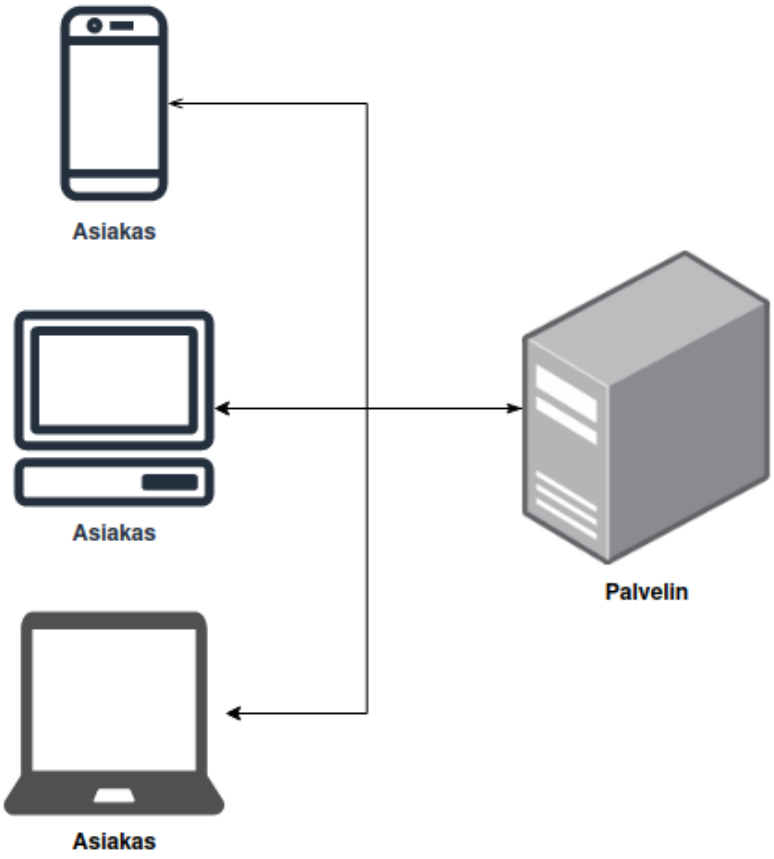
Jacobson, Brail ja Woods (2018) kuvaavat rajapintojen olevan modernin sovelluskehityksen keskiössä. Rajapinnat mahdollistavat useita eri asioita niin liiketoiminnallisesti kuin teknisesti. Rajapintaintegraatioiden kautta alustat ja sovellukset pystyvät kommunikoimaan keskenään ja hyödyntämään rajapintojen tarjoamia

tietovarantoja sekä palveluita. Valmiit rajapintaratkaisut säästävät aikaa ja parantavat tuottavuutta, joka mahdollistaa yrityksiä keskittymään ydintoimintaansa, jolla erottuvat markkinoilla. Rajapinnat luovat myös kokonaan uusia liiketoimintamahdollisuuksia rajapintojen kaupallistamisella. Yritykset voivat myydä pääsyoikeutta rajapintojensa takana oleviin tietolähteisiin ja palveluihin. Tällöin rajapinnan tarjoajan ja rajapinnan kuluttajan välille syntyy molempia hyödyttävä suhde. Tämä luo edellytyksiä rajapintaperusteisiin yhteistyökumppanuuksiin. Tällaista rajapintamonetisaatioon perustuvaa liiketoimintaa kutsutaan myös rajapintataloudeksi. (IBM n.d.) Esimerkkejä rajapintatalouden liiketoimintamalleista ovat muun muassa viestintätyökaluja tarjoava Twilio ja maksuratkaisupalveluita tarjoava Stripe (Gough, Bryant & Auburn 2022).

Rajapintojen ympärille luodaan jatkuvasti palveluita ja liiketoimintoja, jotka ovat tiukasti sidoksissa toisiinsa. Nämä verkostot ja ekosysteemit tukeutuvat toisiinsa ja ovat rajapintaintegraatioidensa varassa. Rajapinnat ovat siis tietynlaisia sopimuksia tarjoajan ja kuluttajan välillä, jossa toinen osapuoli tarjoaa rajapinnan kautta resursseja ja toinen osapuoli on valmis näitä resursseja käyttämään – mahdollista kulua vastaan – odottaen, että rajapinta toimii nyt ja tulevaisuudessa sekä mahdolliset muutokset rajapinnan toiminnassa ja sen tarjoamissa resursseissa on tiedossa. (Ponelat & Rosenstock 2022.)

2.1.1. HTTP

HTTP on tiedonsiirtoon tarkoitettu kommunikaatioprotokolla, jonka toimintaan World Wide Web perustuu. Useimmiten tämä tarkoittaa selaimen ja verkkopalvelimen välistä kommunikaatiota, mutta asiakas-palvelinmallin mukaisesti verkkopalvelimen asiakkaana voi olla myös esimerkiksi laitteet ja mobiilisovellukset. (Pollard 2019). Asiakas-palvelinmalli on tietojenkäsittelyssä käytetty arkkitehtuurirakenne kuvaamaan tietokoneiden ja niiden sovellusten välisiä rooleja verkossa. Osa tietokoneista toimii resurssien hallitsijoina ja tarjoajina, eli palvelimina, kun taas osa tietokoneista toimii niin sanotusti asiakkaina käyttäen palvelimien tarjoamia resursseja. Tämä malli mahdollistaa sen, että yksittäinen palvelin pystyy keskitetysti palvelemaan useampia asiakkaita. (Waterloon yliopisto n.d).



KUVIO 1. Asiakas-palvelinmalli. Kuvassa on kolme eri laitetta kuvaamaan erityyppisiä asiakkaita. Vasemmalla on puhelin, pöytäkone ja kannettava tietokone, oikealla on tietokoneen kuva tietokoneesta kuvaamassa palvelinta. Näiden laitteiden välillä on nuolia, jotka kuvaavat asiakkaiden ja näitä palvelevan palvelimen välisiä yhteyksiä.

Verkkopalvelimet ovat palvelimia eli tietokoneita, jotka ymmärtävät HTTP-protokollaa siihen tarkoitetun sovelluksen avulla, joka pyörii palvelimella, ja tämän myötä pystyvät kommunikoimaan asiakassovelluksen kanssa. Tällaista palvelinta kutsutaankin usein myös HTTP-palvelimeksi. (MDN 2023). Netcraftin (2023) teettämän laajan kyselyn, joka sisälsi vastauksia yli 12 miljoonasta verkkoon liitetystä tietokoneesta, mukaan kaksi käytetyintä HTTP-palvelinsovellusta ovat tällä hetkellä Nginx ja Apache HTTP Server yhteensä 44,6 prosentin markkinaosuudella HTTP-palvelinsovelluksista.

HTTP-palvelin voi tarjota asiakassovelluksille useita erityyppisiä resursseja: HTML-tiedostoja, tyylitiedostoja, kuvia, JSON-dataa, ja niin edelleen (Gourley, Totty, Sayer, Aggarwal & Reddy 2002). HTTP-protokollan ensimmäinen versio, HTTP/0.9, joka kehitettiin vuonna 1991 ja oli alkupiste World Wide Webille, ei tukenut vielä muita tiedostomuotoja kuin HTML-tiedostoja. HTTP-protokollaa on kehitetty ensimmäisestä versiosta paljon eteenpäin, myös tuettujen resurssien osalta. (Grigorik 2017). W3Techsin (2023) mukaan HTTP/2 on tällä hetkellä käytetyin HTTP-protokollan versio. Internet-protokollien standardoinnista vastaava organisaatio IETF julkaisi viime vuonna uusimman version HTTP-protokollasta, HTTP/3 (Bishop 2022). HTTP-protokollaversioiden eroja ei käydä läpi laajasti tässä opinnäytetyössä, ellei protokollaversioiden eroavaisuuksia ole oleellista mainita myöhemmissä kohdissa.

HTTP-palvelimet tarjoavat resursseja asiakassovelluksille palvelimelle lähetettyjen HTTP-pyyntöjen kautta. Sekä asiakkaan lähettämä pyyntö että palvelimen vastaus ovat standardoituja HTTP-viestejä, joiden rakenne on samanlainen riippumatta osapuolesta (kuva 2).

```
HTTP-message   = start-line CRLF
                 *( field-line CRLF )
                 CRLF
                 [ message-body ]
```

KUVA 2. HTTP-viestin rakenne. Kuvassa on yhtälö, joka kuvaa, että mitä HTTP-viesti yleisesti sisältää. HTTP-viesti kuvataan sisältävän rivinvaihtojen lisäksi kolme osaa: aloitusrivi, vaihteleva määrä otsakerivejä ja lopuksi viestirivi. (Fielding, Nottingham & Reschke 2022a.)

Viesti siis alkaa niin kutsulla aloitusrivillä, jota lähettäjän tapauksessa kuvataan myös pyyntörivinä ja vastaajan kohdalla tila- tai vastausrivinä. Tämän jälkeen tulee useampi rivi valinnaisia otsikkokenttiä ja lopuksi yhden rivinvaihdon jälkeen valinnainen viestisisältö. (Fielding ym. 2022a). Seuraavaksi käydään läpi näitä rivejä tarkemmin. Läpikäynti aloitetaan HTTP-pyynnöstä, jonka jälkeen HTTP-viestiä tarkastellaan palvelimen vastauksen näkökulmasta.

HTTP-pyyntö ja pyyntörivi alkaa HTTP-metodilla, joka määrittää, mitä HTTP-pyyntö haluaa tehdä pyynnön kohteena olevalla verkkopalvelimella. Metodeja on yhdeksän kappaletta: GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS, CONNECT, TRACE. Näistä yleisimmät ovat GET- ja POST-pyynnöt, joista GET nimensä mukaisesti hakee resurssin palvelimelta ja POST luo uuden resurssin. Verkkopalvelimen ei tarvitse kuitenkaan tukea jokaista HTTP-metodia. (Gourley ym. 2002). Esimerkiksi Nginx-verkkopalvelimelle tulee määritellä erikseen kaikki tuetut metodit, jolloin määrittelemättömäksi jäänyt HTTP-metodi estetään palvelimelta (Nginx n.d.). Verkkopalvelimen tuetessa OPTIONS-metodia, asiakassovellus pystyy tarkistamaan verkkopalvelimelta, mitä metodeja kyseinen palvelin tukee (Gourley ym. 2002). Jokaisen yhdeksän HTTP-metodien toiminta on avattu liitteessä 1.

Pyyntörivin seuraava osa on URL-osoite. HTTP-pyyntö kohdistuu aina haetun resurssin URL-osoitteeseen palvelimella. URL-osoite koostuu HTTP-pyyntöjen yhteydessä HTTP-protokollan sisältävästä skeemasta, verkkopalvelimen yksilöivästä IP-osoitteesta tai verkkotunnuksesta, ja portista, resurssin relatiivisesta polusta palvelimella sekä mahdollisista avain-arvopareista koostuvista kyselyparametreista. (Gourley ym. 2002).

HTTP-syntaksin ensimmäisen rivin viimeinen osa on HTTP-protokollan versio. HTTP-lähetyksen osapuolet pystyvät tällä versionumerolla kommunikoimaan, mitä HTTP-versiota osapuoli noudattaa ja toinen osapuoli tulee tämän informaation pohjalta toimia yhteensopivasti. (Gourley ym. 2002).

Ensimmäisen HTTP-syntaksirivin jälkeen HTTP-metodin, verkkopalvelimen osoitteen ja HTTP-protokollaversion lisäksi HTTP-pyynnössä lähetetään lähes aina mukana myös otsikkokenttiä, jotka kertovat verkkopalvelimelle tietoja pyynnöstä itsestään sekä pyynnön lähettäneestä asiakassovelluksesta, ja ohjaavat verkkopalvelinta vastaamaan oikealla tavalla lähetettyyn kutsuun. (Gourley ym. 2002). Ainoa pakollinen otsikkokenttä on Host-kenttä, joka kertoo verkkopalvelimelle, kuka pyynnön lähetti. Kenttä on tärkeä sen takia, että palvelin osaa lähettää vastauksen takaisin oikeaan osoitteeseen, koska verkkopalvelimet usein palvelevat useita eri asiakkaita samaan aikaan. Muita otsikkokenttiä ovat esimerkiksi

Accept, joka kertoo palvelimelle, mitä mediatyyppiä asiakas odottaa takaisin palvelimelta, ja User-Agent, joka kertoo palvelimelle yleisellä tasolla, millainen asiakassovellus lähetti HTTP-pyynnön. Internet Assigned Numbers Authority eli IANA ylläpitää listaa rekisteröidystä HTTP-otsikkokentistä. (Fielding ym. 2022a). Mediatyypit, joita kutsutaan myös MIME-tyypeiksi, ovat HTTP-lähetyksen osapuolten tapa kertoa toisilleen HTTP-pyyntöjen sekä HTTP-vastausten sisältötyypistä. HTTP-protokollassa mediatyyppi lisätään Accept- ja Content-Type-otsikkokenttiin. (Fielding ym. 2022b). Mediatyyppi koostuu päätyypistä ja alatyypistä. Päätyyppejä on seitsemän kappaletta: text, image, audio, video, application, multipart ja message. (Freed & Borenstein 1996.) Verkkoselaimen palauttaessa esimerkiksi HTML-sivun käyttäjälle GET-pyynnön vastatessa, on tällaisessa HTTP-vastauksessa yleensä otsikkokenttä Content-Type ja sen arvona text/html.

HTTP-pyyntö voi sisältää otsikkokenttien ja rivinvaihdon jälkeen vielä lopuksi erillisen viestisisällön raakadatana, jos pyynnön metodi tukee sitä. Tällaisia metodeja ovat resurssia luovat ja muokkaavat POST-, PUT- ja PATCH-metodit. HTTP-pyynnön viestisisältö voi sisältää useaa erityyppistä dataa, kuten verkkopalvelimen tarjotessa resursseja asiakassovelluksille. Sisältö mahdollisesti koodataan ja lähetetään verkkopalvelimelle, joka käsittelee HTTP-pyynnön mukaan lukien pyynnön viestisisällön. Viestisisältö lähetetään usein yhtenä data, mutta viestisisältö voidaan lähettää myös useammassa osassa. Useimmiten tällainen moniosainen viestisisältö on lomakkeen kautta tullutta kenttädataa. HTTP-pyynnön sisältäessä viestisisällön, myös HTTP-pyyntöön lisätään otsikkokentäksi Content-Type kuvaamaan lähetetyn sisällön tyyppiä. Esimerkiksi moniosaisen lomakedatan mediatyyppi voi olla multipart/form-data. (Gourley ym. 2002). HTTP-pyynnön URL-osoitteessa voi lähettää myös kyselyparametreja, mitkä verkkopalvelin pystyy käsittelemään ylimääräisenä tietona HTTP-pyynnön yhteydessä. Kyselyparameterit toimivat siis samankaltaisesti kuin viestisisältö. Olennainen ero näiden kahden tavan välillä löytyy kuitenkin tietoturvasta. Kyselyparametrit näkyvät selaimessa, selaimen historiassa sekä URL-osoitteita jakaessa, joka on ongelmallista käyttäjän lähettäessä arkaluonteista dataa, kuten salasanoja. Viestisisältö puolestaan ei jää talteen selaimeen ja on täten turvallisempi tapa lähettää dataa verkkopalvelimelle. (Pollard 2019).

Määritelty HTTP-pyyntö lähetetään verkkopalvelimelle, joka otetaan vastaan, käsitellään ja siihen myös vastataan. HTTP-vastaus noudattaa samaa rakennetta kuin mitä kuvassa 1 esitetään, mutta tilarivin sisältö on hieman erilainen. Rivin ensimmäisenä osa on HTTP-protokollaversio, joka oli HTTP-pyynnön pyyntörivin viimeinen osa. Toinen ja kolmas osa esiintyvät vain HTTP-vastauksen tilarivillä: vastauksen tilakoodi ja sen tekstimuotoinen selite. (Fielding ym. 2022a). HTTP-pyynnön tehnyt asiakassovellus tulee tietää, mitä pyynnön kanssa tapahtui, jotta asiakassovellus osaa reagoida verkkopalvelimen vastaukseen vaaditulla tavalla. HTTP-pyynnön kanssa voi käydä useita eri asioita, joita kolminumeroiset tilakoodit kuvaavat. Tilakoodit on määritelty viiteen eri kategoriaan kuvan 2 mukaisesti. (Gourley ym. 2002).

Overall range	Defined range	Category
100-199	100-101	Informational
200-299	200-206	Successful
300-399	300-305	Redirection
400-499	400-415	Client error
500-599	500-505	Server error

KUVA 3. HTTP-vastausten tilakoodikategoriat. Kuviossa on taulukko, jossa on kolme saraketta: tilakoodinumeroiden arvojoukko, tosiasiallinen arvojoukko, joita käytetään tilakoodeihin, tilakoodijoukon kategoria. Tilakategorioita on viisi kappaletta: informationaaliset, onnistuneet, uudelleenohjatut, asiakaspään virheet ja palvelinpään virheet. (Gourley ym. 2002.)

On siis informatiivisia, onnistumista kuvaavia, uudelleenohjausta kuvaavia, asiakassovelluksen virheen kuvaavia ja verkkopalvelimen virheen kuvaavia tilakoodeja. Näemme myös kuvasta, että vaikka kategorioille on varattu sadan luvun numerosarja, on määriteltyjä tilakoodeja kuitenkin paljon vähemmän kuin mitä varattuja koodeja. Esimerkkejä yleisistä tilakoodeista ovat 200, joka kuvastaa onnistunutta HTTP-kutsua, 401, joka kertoo valtuuksien, kuten käyttäjätunnuksen ja salasanan puuttumisesta, ja 404, joka ilmoittaa, että verkkopalvelin ei löytänyt pyydettyä resurssia. Kolminumeroisten tilakoodien kanssa tulee myös tekstimuotoinen seliteteksti siitä, mitä pyynnön kanssa tapahtui, joka sopii paremmin ihmisluettavaksi syyksi. (Gourley ym. 2002). Määritelty lista tilakoodeista ja niiden ihmisluettavista seliteteksteistä löytyy liitteestä 2.

HTTP-pyyntöjen toimintaa asiakassovelluksena voi testata esimerkiksi verkko-ohjelma Netcatilla, jolla voi lähettää komentoja ja dataa tekstipohjaisesti TCP-yhteyksien yli. Jos olisi tarvetta selvittää, millaisia otsakkeita palvelin palauttaa HTTP-vastauksen mukana, voi tähän tarkoitukseen käyttää yksinkertaista HEAD-pyyntöä. HEAD on lähes identtinen GET-pyynnön kanssa, mutta HTTP-palvelin ei HEAD-pyyntöön vastatessa palauta resurssin sisältöä vaan pelkästään resurssiin liittyvät otsikkokentät. Tämä voisi Netcatilla näyttää seuraavalta:

```
$ netcat example.com 80
HEAD / HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
Accept-Ranges: bytes
Age: 489101
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Wed, 20 Sep 2023 20:19:04 GMT
Etag: "3147526947+gzip"
Expires: Wed, 27 Sep 2023 20:19:04 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Server: ECS (bsb/27DC)
X-Cache: HIT
Content-Length: 1256
```

KUVA 4. HTTP-pyynnön tekeminen Netcat-työkalulla. Kuva on otettu terminaalista, joka alkaa netcat-työkalun kutsumisella osoitteeseen example.com ja porttiin 80. Tämän jälkeen näytetään tehtävän kutsun sekä kutsun vastauksen HTTP-tiedot otsakkeineen.

Huomaamme kuvasta 4, että netcat-komento sai muodostettua yhteyden example.com-verkkotunnuksen HTTP-palvelimeen, jonka jälkeen kaksi seuraavaa riviä määrittää palvelimelle lähetetyn HTTP-pyynnön, ja lopuksi yhden rivinvaihdon jälkeen palvelin on vastannut takaisin resurssin otsikkokentillä. HTTP-pyyntö ja sen transaktio oli siis onnistunut ja pystymme näin näkemään, millaisia vastauksia palvelin vastaa takaisin.

2.1.2. REST

REST on arkkitehtuurityyli web-pohjaisille verkkosovelluksille. REST määrittelee tietyt rajoitukset, joiden tarkoituksena on ohjata verkkosovellusten suunnittelua ja toteutusta tavalla, joka tukee internetin keskinäisten sovellusten toimivuutta, nopeutta ja skaalautuvuutta. REST on kuvattu ensimmäisen kerran Roy Fieldingin vuonna 2000 tehdyssä väitöskirjassa. Roy Fielding oli tätä ennen luomassa HTTP-protokollan kahta ensimmäistä versiota, ja tätä taustaa vasten Fielding halusi luoda standardin, joka vastaa HTTP-protokollalla toimivan internetin vaatimuksiin ja tarjoaa hyviä ohjenuoria sovellusten toteuttajille. (Fielding & Taylor, 2020.)

REST koostuu kuudesta eri rajoituksesta, jotka toteutuessaan kuvaavat REST-arkkitehtuurityylin mukaisen verkkosovelluksen ja sen rajapinnan (Fielding 2000).

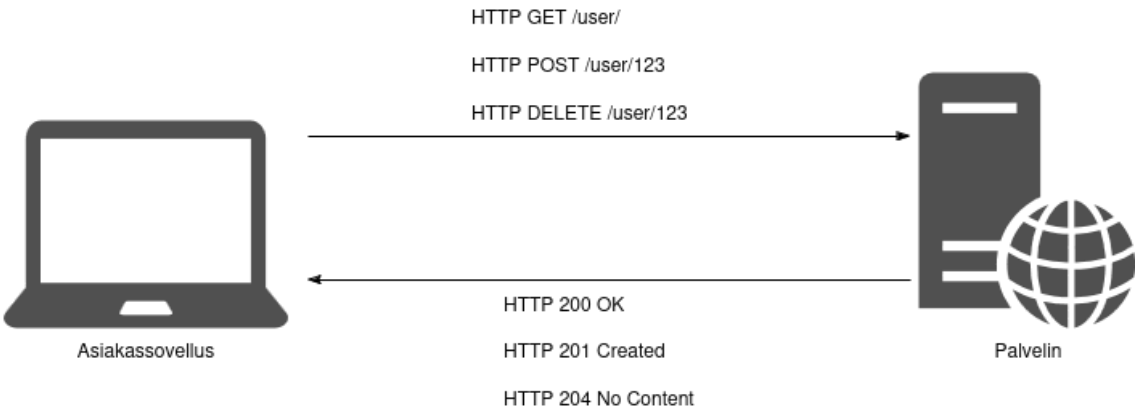
Ensimmäinen näistä rajoituksista on asiakas-palvelin-arkkitehtuurimalli. Tämän rajoituksen tarkoituksena on vähentää asiakassovelluksen ja palvelimen keskinäistä riippuvuutta toisistaan. Asiakassovelluksen ei tarvitse käyttöliittymänä toimia tiedon ja tilan tallentajana vaan tämä vastuu tulisi olla palvelimella. Fielding mainitsee suurimpana hyötynä sen, että palvelimen toiminta voi kehittyä ja muuttua vapaasti ilman vaikutusta asiakassovelluksen toimintaan (Fielding 2020).

Vahva riippuvuus asiakassovelluksen ja palvelimen välillä voi tarkoittaa sitä, että muutos palvelimella johtaa muutoksiin myös asiakassovelluksen puolella.

Toinen rajoitus on lisärajoite ensimmäisen rajoitteelle eli asiakas-palvelin-arkkitehtuurimallille. Asiakas-palvelinmallin toteutus tulisi olla tilaton. Tämä tarkoittaa sitä, että palvelimen ei tarvitse pitää lisäkontekstia tallessa, jotta vastaus palvelimelle tulevaan HTTP-pyyntöön on mahdollinen. Kaikki tarvittava tieto pyynnön toteuttamiseen tulisi olla siis pyynnössä itsessään mukana. Tämä tarkoittaa Fieldingin mukaan sitä, että myös sovelluksen istuntotila tulisi olla tallennettuna asiakassovelluksen puolella. Tämän rajoituksen tarkoituksena on yksinkertaistaa palvelimen rakennetta ja parantaa pyyntöjen sisällön n, järjestelmän luotettavuutta, virheiden sietokykyä, monitoroitavuutta ja skaalautuvuutta. Tilattomuuden rajoitteen tuomien positiivisten vaikutusten ohella Fielding toteaa tilattomuuden olevan kuitenkin kompromissi, koska sillä on negatiivisia vaikutuksia esimerkiksi verkon suorituskykyyn palvelimelle lähetettyjen pyyntöjen kontekstitiedon takia ja palvelun johdonmukaiseen käyttäytymiseen, jos asiakassovellukset ovat implementoineet istuntotilan hallinnan komponentteja eri tavalla. (Fielding 2020.) Yksi käytetyimmistä ratkaisuista nykypäivänä tilattomaan palvelinkommunikaatioon on JSON Web Tokenit, jotka mahdollistavat JSON-pohjaisen merkkijonon lisäämisen HTTP-pyynnön auktorisointiotsakkeeseen, jonka avulla palvelin voi autentikoida ja auktorisoida käyttäjän rajapintaan tulevan kutsun yhteydessä (Haro 2023).

Kolmantena rajoitteena sovellusten tulisi käyttää välimuistia mahdollisuuksien mukaan verkon tehokkuuden parantamiseksi ja latausaikojen pienentämiseksi (Fielding 2000). Välimuisti tarjoaa nimensä mukaisesti väliaikaisen muistin asiakassovelluksen ja palvelimen väliin, johon voidaan tallettaa haettuja resursseja lähemmäksi asiakassovellusta, jotta resurssia ei tarvitse hakea palvelimelta asti uudestaan, jos käyttäjän pyynnössä ei ole tapahtunut muutoksia. Jos verkkosivulla käytettävän tiedon hakemiseen joutuu tekemään raskasta laskentaa, välimuistin käyttäminen voi nopeuttaa sivuston latautumista käyttäjille huomattavasti. Välimuisti voi sijaita useammassa eri paikassa. Selaimiin on sisäänrakennettu oma välimuisti, joka on helposti käytettävissä, mutta tämän lisäksi sovellusten järjestelmiin voidaan rakentaa myös välityspalvelin, joka toimii välimuistina. (Amazon Web Services n.d..)

Neljäs ja hyvin keskeinen rajoite on REST-rajapintojen yhtenäinen toteutus, jonka pyrkimyksenä on yksinkertaistaa järjestelmäarkkitehtuuria implementoinnin ja ymmärrettävyyden vuoksi sekä parantaa asiakassovelluksen ja palvelimen välisen kommunikaation näkyvyyttä (Fielding 2000). Tämän yleisen rajoitteen lisäksi REST-rajapinta tulisi noudattaa rajapintaimplementaation neljää lisärajoitetta: rajapinta identifioi resurssit uniikisti URI-polkujen avulla, resursseja ei haeta ja manipuloida suoraan vaan asiakassovellus voi pyytää palvelimelta esitystapoja resursseista erilaisten serialisointiformaattien, kuten HTML- ja JSON-formaattien, avulla rajapinnan kautta, kommunikaatio palvelimen kanssa on tilaton toisen rajoitteen mukaisesti ja hypermedia toimii sovelluksen tilakoneena (Haro 2023). Näistä viimeisin tarkoittaa sitä, että rajapinnan tarjoama resurssin esitystapa sisältää mukanaan linkkejä, joilla käyttäjää voidaan johdattaa sovelluksen eri tiloihin ja toimintoihin, jotta rajapinta olisi navigoitavissa sekä helposti käytettävä ja ymmärrettävä (Kotilainen 2009, 12).



KUVIO 5. REST-rajapinnan resurssien tunnistaminen URI-polkujen avulla HTTP-pyynnössä. Kuvassa on palvelimen kuva ja asiakassovelluksen kuva. Näiden välillä kulkee viivat. Palvelimelle menevän viivan päällä on palvelimelle meneviä GET-, POST- ja DELETE-kutsuja, ja asiakassovellukselle menevän viivan alla on palvelimen 200-, 201- ja 204-vastaukset kutsuihin.

Viides rajoite ohjaa rakentamaan järjestelmän kerrosarkkitehtuurillisesti. Järjestelmän kerroksellisuus rakentuu siten, että asiakassovelluksen ja vastaanottavan palvelimen välillä voi olla moniakin välikomponentteja ja -ohjelmistoja, jotka voivat käsitellä lähetettyä kutsua. Koko järjestelmä koostuu siis asiakassovelluksen,

palvelimen ja näiden väliin asetettavien komponenttien yhteistoiminnasta. Kerroksellisen järjestelmän komponenttien toiminta tulee kuitenkin olla rajattua vain omaan kerrokseensa, jotta tietyn kerroksen toteuttava komponentti voidaan vaihtaa vaivattomasti toiseen tarvittaessa eikä kerrosten yli vaikuttavat riippuvuudet vaikuta tähän ja tuo järjestelmään lisää kompleksisuutta. (Fielding 2000.) Kerrosarkkitehtuurin välikerrokset voidaan toteuttaa esimerkiksi välityspalvelimina ja yhdyskäytävinä (Masse 2011).

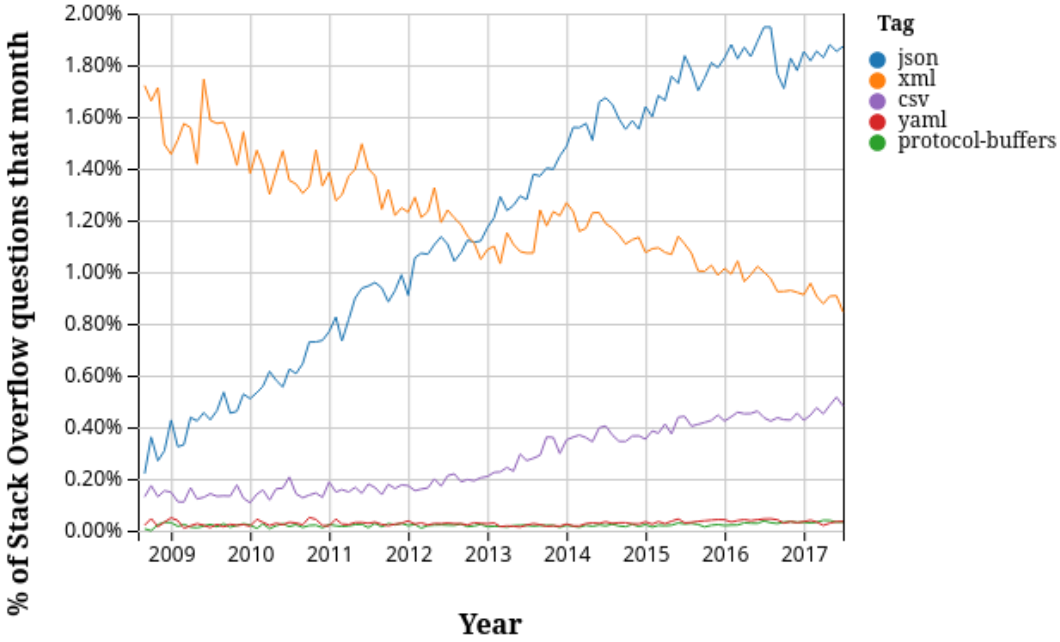
Kuudes ja viimeinen rajoite, joka on myös rajoitteista ainoa valinnainen rajoite, on mahdollisuus ladata koodia, jonka avulla asiakassovellus pystyy lisäämään ominaisuuksia järjestelmään dynaamisesti. Tämä parantaa järjestelmän tehokkuutta, sillä osa järjestelmän logiikasta ja laskennasta siirtyy palvelimelta asiakassovellukselle, ja järjestelmän laajennettavuutta. Tämä kuitenkin vähentää näkyvyyttä monitorointiin, sillä tieto ei kulje enää palvelimen ja asiakassovelluksen välillä. (Fielding 2000).

2.2. OpenAPI-spesifikaatio

OpenAPI-spesifikaatio (OAS), on rajapintojen kuvausformaatti REST-pohjaisten HTTP-rajapintojen kuvaamiseen. OpenAPI-spesifikaatio määrittelee spesifikaatiossaan rakenteen, millä tavoin rajapinnat kuvataan. Rajapintakuvausten tarkoituksena on tarjota tapa ylläpitää ja jakaa tietoa rajapinnoista eri käyttötapauksia ja työkaluja varten, jotka tukevat rajapintojen hallintaa ja ylläpitoa rajapintojen koko elinkaaren ajan. (OpenAPI Initiative, kun.d.)

Alunperin OAS tunnettiin Swagger-projektina. Swagger oli Wordnik-nimisessä yrityksessä vuonna 2010 aloitettu projekti, jonka tarkoituksena oli kuvata rajapintoja JSON-muotoisessa formaatissa. Swagger syntyi aikoinaan tilanteessa, jossa pitkään vallitsevana tiedonsiirtoformaattina toimi XML ja XML-pohjaiset REST- ja SOAP-rajapinnat olivat vakiintuneita tapoja palvelinten kanssa kommunikoitaessa. XML-formaattia käytettiin myös rajapintojen kuvaamiseen. REST-rajapintoja kuvattiin WADL-formaatilla ja SOAP-rajapintoja kuvattiin WSDL-formaatilla. (Gardiner 2018.)

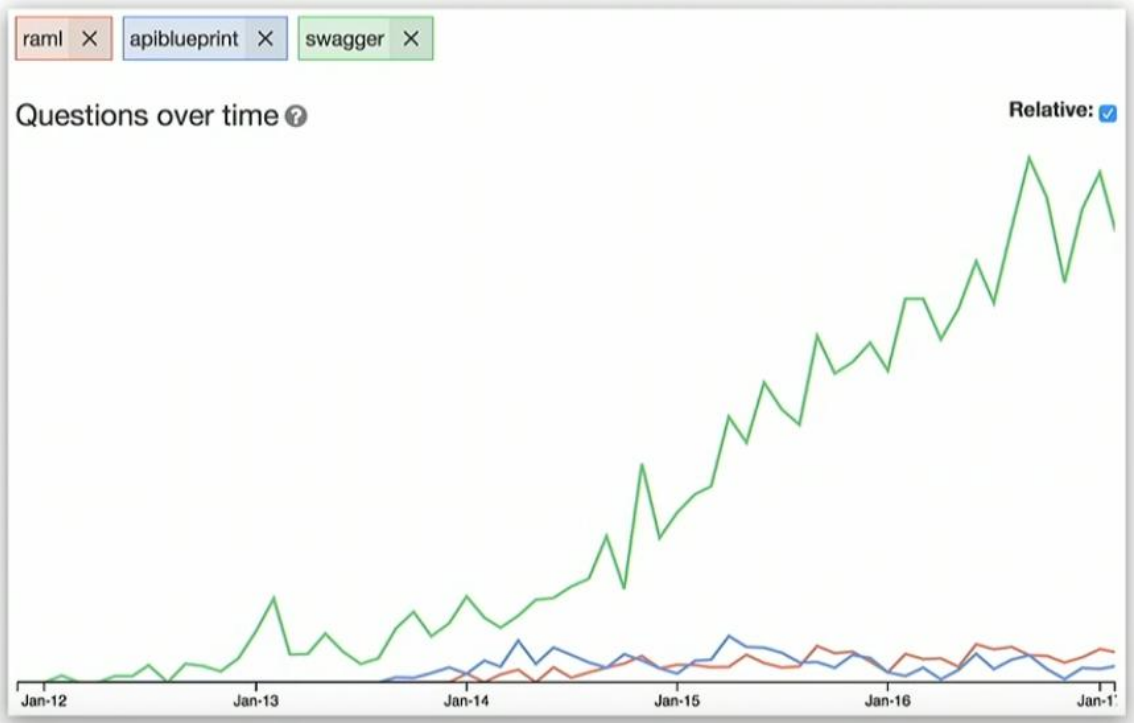
Samoihin aikoihin nähtiin myös kasvua JSON-pohjaisten REST-rajapintojen käytössä. (Gardiner 2018). JSON-tiedonsiirtoformaatti sai alkunsa jo vuonna 2001, mutta formaattina sen suosio alkoi kasvamaan vasta 2000- ja 2010-luvun vaiheessa (Target 2017).



KUVIO 6. Kaavio tiedonsiirtoformaatteihin liittyvien Stack Overflow -kysymysten suhteellisten osuuksien kehityksestä. Kaaviossa esiintyy JSON-, XML-, CSV-, YAML- ja Protocol Buffers -tiedonsiirtoformaatit. Kaavion Y-akseli kuvaa Stack Overflow -kysymysten suhteellista osuutta kuukausittain asteikolla 0.00 % - 2.00 % ja X-akseli kuvaa vuosilukuja vuosien 2009 ja 2017 välillä. JSON kuvataan selkeästi kasvaneimpana tiedonsiirtoformaattina. (Target 2017.)

JSON-pohjaisia rajapintoja kuitenkin kuvattiin samoilla XML-pohjaisilla rajapintojen kuvaukseen tarkoitetuilla työkaluilla. Tämä kuitenkin todettiin huonoksi tavaksi. Samoihin aikoihin Wordnik alkoi kehittämään Swaggeria JSON-pohjaisten rajapintojensa kuvaamiseen ja dokumentointiin. Swagger näytti toimivan ratkaisuna näihin haasteihiin, mitä XML:n kanssa toimiessa koettiin. Swagger huomattiin myös Apigee-nimisen yrityksen toimesta, jonka johdosta alkoivat keskustelut Swaggerin Creative Commons -lisenssoinnista ja Swaggerin julkaisusta avoimena lähdekoodina. Swaggerin kehitys jatkui seuraavina vuosina, mutta Swaggerissa oli vielä selkeitä puutteita. Tämän vuoksi Swaggerin seuraavaa versiota

lähdettiin kehittämään, ja versio 2.0 julkaistiin vuonna 2014. Samana vuonna perustettiin Swagger Working Group edistämään Swaggerin kehitystä. (Gardiner 2018.) Swagger oli jo aikaseimminkin nähnyt käyttöä, mutta odotetun 2.0-versiojulkistuksen sekä Swagger Working Groupin perustamisen myötä kiinnostus alkoi entisestään kasvamaan (kuvio 7).



KUVIO 7. Kaavio Swaggerin ja kahden muun rajapintakuvausformaatin välisistä suhteellisista Stack Overflow -kysymysten lukumäärien kehityksestä. Y-akseli kuvaa ilman eksplisiittistä asteikkoa suhteellista kasvua ja X-akselia vuotta vuosien 2012 ja 2017 välillä. Swagger on ainoa kolmesta vaihtoehdosta, joka on kasvanut. (Gardiner 2018.)

Vuoden 2015 keväällä Wordnik myi Swagger-projektinsa SmartBearille. Swaggerista toivottiin käytön kasvun yhteydessä alan yhteistä standardia, joten Swaggerin myyminen toiselle kaupalliselle toimijalle nähtiin uhkana. Viimeistään tässä kohtaa nähtiin tärkeänä, että Swagger siirtyisi puolueettoman, hallinnollisen organisaation alle, jolla ei olisi kaupallisia tavoitteita. (Gardiner 2018.) Tämän takia vuoden 2015 lopulla Linux Foundation -säätiön tukemana ja alaisuuteen perustettiin kymmenen toimijan, joihin kuuluivat alan suurimpia yrityksiä, kuten Google, IBM ja Microsoft, OpenAPI Initiative -projekti, jolle SmartBear lahjoitti Swagger-spesifikaation. Vuoden 2016 ensimmäisestä päivästä lähtien spesifikaatio alkoi

kulkea nimellä OpenAPI Specification (OpenAPI Initiative n.d.). Swaggerin siirtyminen OpenAPI Initiativen hallintaan kasvatti formaatin käyttömääriä entisestä nopeammin (kuvio 7).

Vuonna 2017 julkaistiin OpenAPI Specification -formaatin versio 3.0, ja vuonna 2021 julkaistiin versio 3.1, joka on nykyisin versio (OpenAPI Initiative n.d.).

Tämän opinnäytetyön puitteissa keskitytään pääosin vain uusimpaan versioon OpenAPI Specification -formaatista ellei aiempien versioista ole oleellista keskustella kontekstin vuoksi.

2.2.1. Rajapinnan kuvaaminen OpenAPI-spesifikaatiolla

Rajapinnan tehtävänä on määritellä, mitä rajapinnan takana oleva palvelu pystyy tarjoamaan rajapinnan käyttäjälle sekä miten palvelun kanssa kommunikoidaan ja toimitaan. Rajapinta on siis sopimus ja lupaus käyttäjälle palvelusta. Muut toimijat pystyvät tämän sopimuksen pohjalta rakentamaan omia palveluitaan hyödyntäen rajapinnan tarjoamia palveluita ja resursseja. Tämän luottamus- ja käyttösuhteen takia rajapintojen käyttäjien tulee pystyä luottamaan siihen, että rajapinnan toiminta pysyy vakaana ja muuttumattomana, ellei muutoksista ilmoiteta erikseen tarpeeksi aikaisin ja selkeästi, jotta toimijat voivat tehdä tarvittavat muutokset palveluihinsa ajoissa ja hallitusti. (Rosenstock & Ponelat 2022.)

OpenAPI Initiative tarjoaa tähän ratkaisuksi standardoidun ja ohjelmointikieliagnostisen rajapintojen kuvausformaatin, OpenAPI-spesifikaation. OpenAPI-spesifikaatio on kuvausformaatiltaan teknisesti JSON-objekti, joka on kuvatusta rajapinnasta tehty skeema eli rakenne (OpenAPI Initiative 2021). OpenAPI-spesifikaation rakenne on täysin yhteensopiva JSON Schema -spesifikaation kanssa. JSON Schema on määrittänyt spesifikaationsa mukaan JSON-rakenteen, jota käytetään ensisijaisesti JSON-objektien validointiin. OpenAPI-spesifikaation kontekstissa JSON Schema määrittelee suurelta osin avainsanat, mitä skeeman rajapintakuvauksissa voidaan käyttää. OpenAPI lisää näiden lisäksi joukkoon omia avainsanoja ja formaattiarvoja. (Desrosiers 2021.)

Rajapinnan OpenAPI-skeema voidaan kuvata joko JSON-formaattina tai YAML-formaattina omassa tiedostossaan. (OpenAPI Initiative 2021.) YAML on data-formaattina JSON-dataformaatin ylijoukko, jonka takia rajapintakuvaukset voidaan kirjoittaa molemmilla formaateilla (YAML 2021). Skeemadokumentin avulla sekä ihmiset että tietokoneet voivat hyödyntää kuvausta rajapinnasta eri käyttötarkoituksia varten (OpenAPI Initiative 2021).

Spesifikaation mukaan OpenAPI-dokumentti täytyy sisältää openapi-kentän, joka ilmoittaa, mitä OpenAPI-spesifikaation versiota dokumentti noudattaa, ja info-kentän, joka sisältää metadataa rajapinnasta. Näiden lisäksi rajapinnan täytyy toteuttaa myös vähintään yksi paths-, webhooks- tai components-kentistä, jotka tarkemmin kuvaavat rajapinnan päätepisteitä. Kaikista yleisin käytetty näistä on paths-kenttä, joka kuvaa rajapinnan päätepisteen. Päätepisteestä kuvataan operaatiot, joita päätepiste tukee. Rajapinnasta voidaan kuvata päätepisteen polulle annettavat hakuparametrit, jotka ovat yhteisiä kaikille operaattoreille tai sitten hakuparametreja voidaan kuvata yksittäisen operaation näkökulmasta. Parametreille voidaan myös antaa skeemakuvaus, joka lisää parametriin rajoituksia esimerkiksi pakollisuuden tai hyväksyttyjen raja-arvojen osalta. Parametrien lisäksi operaatioista voidaan kuvata muun muassa operaatioiden toiminta kuvausteksteillä, operaatioiden vastaanottamien pyyntöjen rungot ja palautettavat vastaukset asiakkaille. OpenAPI-dokumenteissa voidaan käyttää myös uudelleenkäytettäviä JSON-objekteja, joilla voidaan kuvata useita eri osia OpenAPI-dokumenteista, kuten uudelleenkäytettäviä palautettavia vastauksia, parametrimäärittelyjä, pyyntöjen runkomäärittelyjä, ja jopa kokonaisia päätepisteiden polkumäärittelyjä. (OpenAPI Initiative 2021.)

OpenAPI-spesifikaatio tarjoaa myös kuvausmäärittelyjä rajapinnan autentikaation ja auktorisoinnin osalta. Vaikka OpenAPI kuvaa pääasiallisesti JSON-tietoformaatilla käytävää tiedonvaihtoa asiakkaan ja palvelimen välillä, niin OpenAPI-dokumenteilla voidaan kuvata myös XML-dataa, ja tähän löytyy myös oma määritellyobjekti, joka tukee XML-datan kuvaamista. (OpenAPI Initiative 2021.)

Tätä dokumenttia voidaan hyödyntää sekä uusien että olemassa olevien ohjelmistojen kohdalla. Dokumentilla voidaan täydentää olemassa olevan ohjelmiston dokumentaatiota esimerkiksi selkeämmän kommunikoinnin, päätöksenteon ja

ohjelmiston elinkaaren hallinnoinnin tueksi. Vielä suuremman avun dokumentti tuo uuden ohjelmiston suunnittelun ja implementoinnin kohdalla. Uuden ohjelmiston suunnittelun ja implementoinnin edistämiseksi on eri lähestymistapoja. Lähestymistavoissa on eroavaisuuksia siinä, millä tavoin ohjelmistossa määritellään esimerkiksi tietomallit ja rajapinnat. (Tucci 2023.)

Jos ohjelmiston kehitys on vielä prototyyppivaiheessa, ohjelmiston määrittely vaatii vahvaa iterointia, toteutus on vahvasti aikataulurajoitettua, ja toteuttamisen selkeyden vuoksi aloittamiseen ei vaadita vahvaa suunnitteluvaihetta, voi olla perusteltua lähestyä ohjelmiston määrittelyä siten, että sitä lähdetään suoraan toteuttamaan. Tällöin toteuttamisessa voidaan edetä suhteellisen nopeasti ja ohjelmiston osat määräytyvät ja kehittyvät orgaaniseesti toteutusvaiheen aikana. Tämä lähestymistapa voi johtaa kuitenkin dokumentoinnin puutteisiin ja ohjelmiston epäselkeyksiin. (Tucci 2023.)

Näihin haasteisiin voidaan pyrkiä vastaamaan siten, että ennen toteutusvaihetta ohjelmiston arkkitehtuuri, tietomallit ja rajapinnat määritellään huolellisesti, joka tekee ohjelmiston eri osien kehittämisestä yhtenäisempää ja ohjelmiston dokumentoinnista kattavampaa, joka tukee myös tulevaisuudessa tapahtuvaa kehitystä. Tätä lähestymistapaa noudattaessa ohjelmistolla voi olla taustavaatimuksia, jonka takia ohjelmiston pitkäikäisyyteen, vakauteen ja dokumentaatioon tulee kiinnittää erityistä huomiota, jolloin huolellisella suunnitelulla on tärkeämpi rooli. (Tucci 2023.)

Etenkin suunnitteluvetoisessa lähestymismallissa OpenAPI-dokumenttia voidaan käyttää tehokkaasti rajapinnan suunnitteluun, koska sen avulla kaikki rajapinnan osat voidaan määritellä etukäteen. Dokumenttiin voidaan sisällyttää tiedot rajapinnan poluista, tuetuista metodeista, toiminnoista, tietomalleista, ja mahdollisista autentikaatio- ja auktorisointitiedoista. Kun rajapinnan suunnitteluw on täten määritelty ja dokumentoitu huolellisesti OpenAPI-dokumenttiin, voidaan tätä käyttää pohjana kaikkeen muuhun tekemiseen. Kehitystiimin eri osapuolet tietävät jo kehitystyön alkuvaiheessa, millaista tietomallia ja toimintalogiikkaa rajapinta noudattaa, jonka johdosta jokainen osapuoli voi yhteisen rajapintasopimuksen pohjalta aloittaa toteutuksen. Tämä edistää yhtenäisempien ohjelmistoratkaisujen

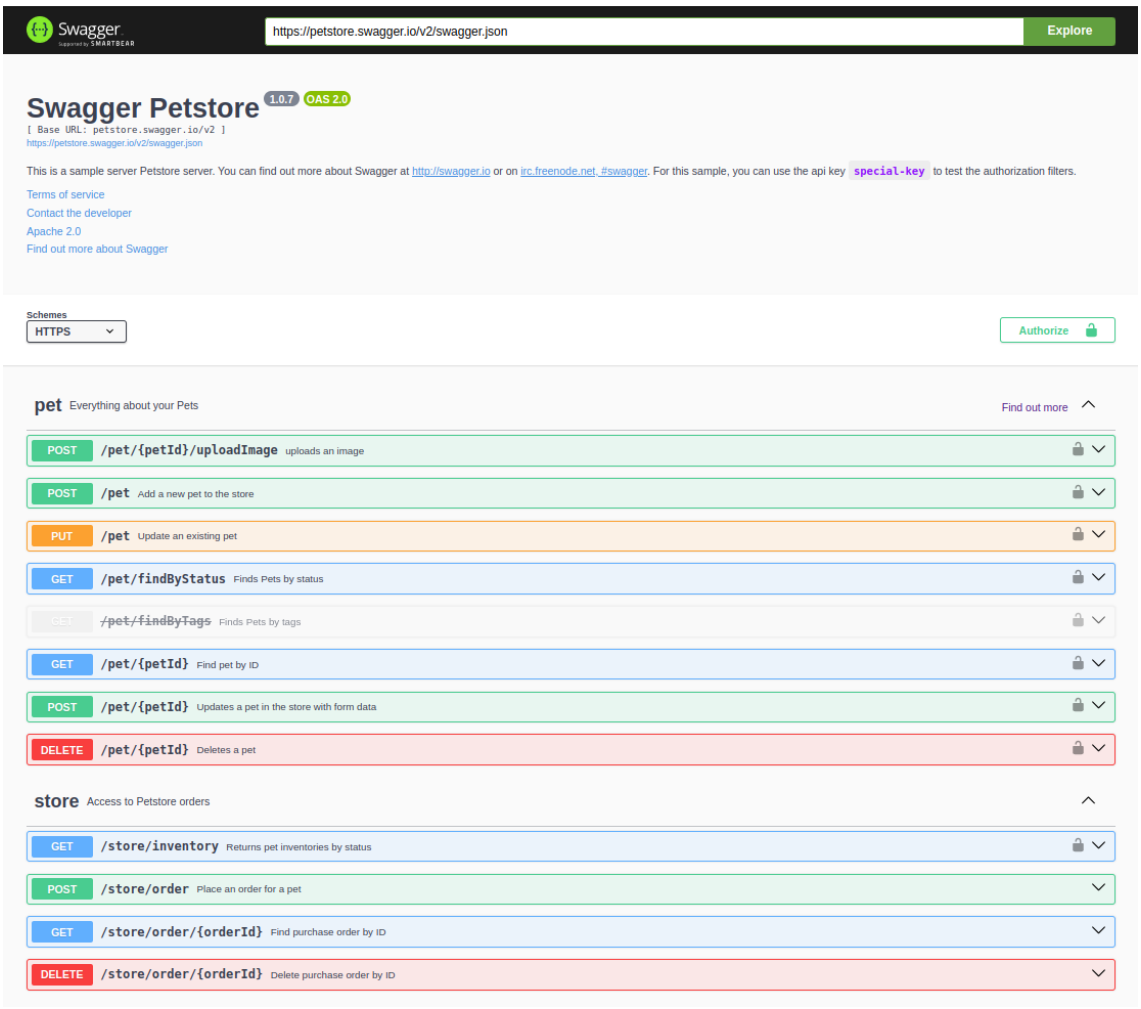
luomista ja samalla OpenAPI-dokumentti toimii myös teknisenä rajapintadokumentaationa kehittäjille. Toteutusvaiheessa esiintyvät muutostarpeet rajapintaan voidaan myös toteuttaa keskitetysti versiohallitun OpenAPI-dokumentin avulla. Toteuttajilla ei tarvitse olla edes pääsyä rajapinnan toteutukseen, koska oleelliset muutokset rajapinnan käyttäjille on dokumentoitu OpenAPI-dokumenttiin. (Tucci 2023.)

OpenAPI-dokumentti tarjoaa myös mahdollisuuden käyttää hyväksi koko OpenAPI-ekosysteemin tarjoamia työkaluja, jotka hyödyntävät OpenAPI-spesifikaatiostandardia, läpi rajapinnan ja sovelluksen elinkaaren. Näitä työkaluja käsitellään alaluvuissa 2.2.2 ja 2.2.3.

2.2.2. OpenAPI-dokumentaation generointi

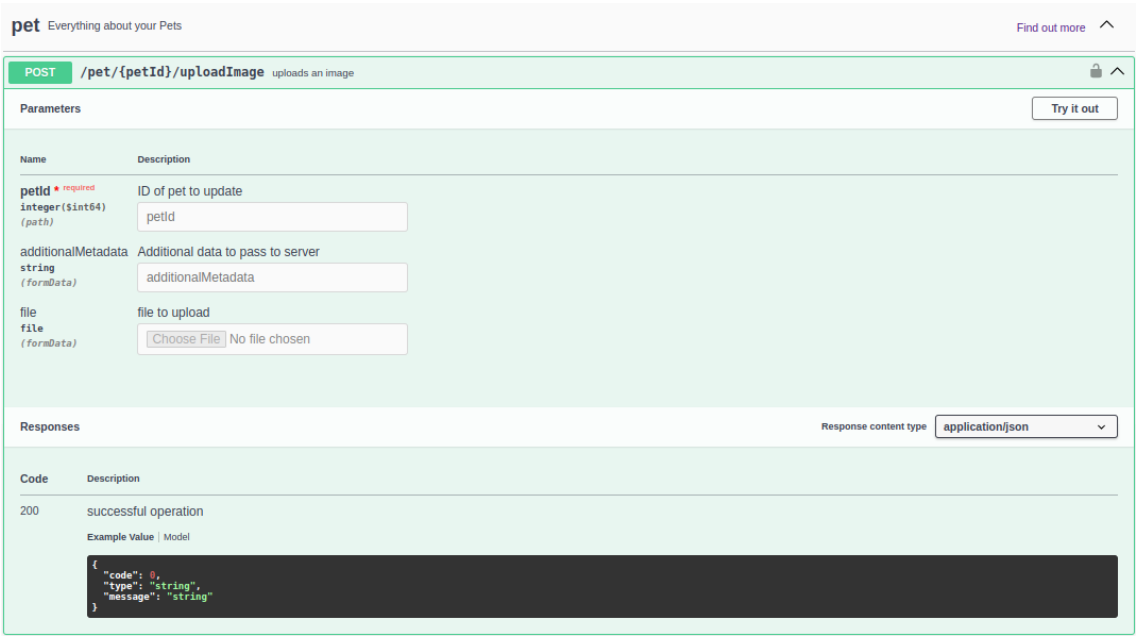
OpenAPI-dokumentti auttaa jo itsessään rajapinnan dokumentoinnin kanssa ja tätä dokumenttia voidaan käyttää kehityksen tukena sellaisenaan, koska se sisältää huolellisesti tehtynä kaikki tarvittavat tiedot rajapinnoista. Mutta OpenAPI-dokumentti on kuitenkin loppujen lopuksi spesifikaatioon perustuva, koneluettava dokumentti, jota voidaan parsia tämän spesifikaation myötä koneellisesti. Tämän takia Swagger on luonut dokumentaatiotyökalun Swagger UI, joka luo OpenAPI-dokumentin pohjalta automatisoidusti visualisoidun ja interaktiivisen käyttöliittymän, joka on enemmän ihmisluettava ja tekee dokumentaation käyttämisestä luontevampaa sekä taustajärjestelmän että käyttöliittymäpuolen implementointiin. (Swagger n.a.) Gardinerin (2018) mukaan Swagger UI on ollut jo pitkään tunnetuin käyttötapaus OpenAPI-dokumentaation käytölle.

Käyttöliittymästä näkee rajapinnan nimen sekä kuvauksen. Rajapinnan jokaisesta polusta näkyy myös kyseisen polun tukemat metodit sekä alipolut. Poluista näytetään myös niiden kuvaukset, mitkä ovat OpenAPI-dokumenttiin kirjoitettu. Rajapinnan käyttämä HTTP-skeema ja auktorisointitiedot ovat myös saatavilla. (Kuva 8.)



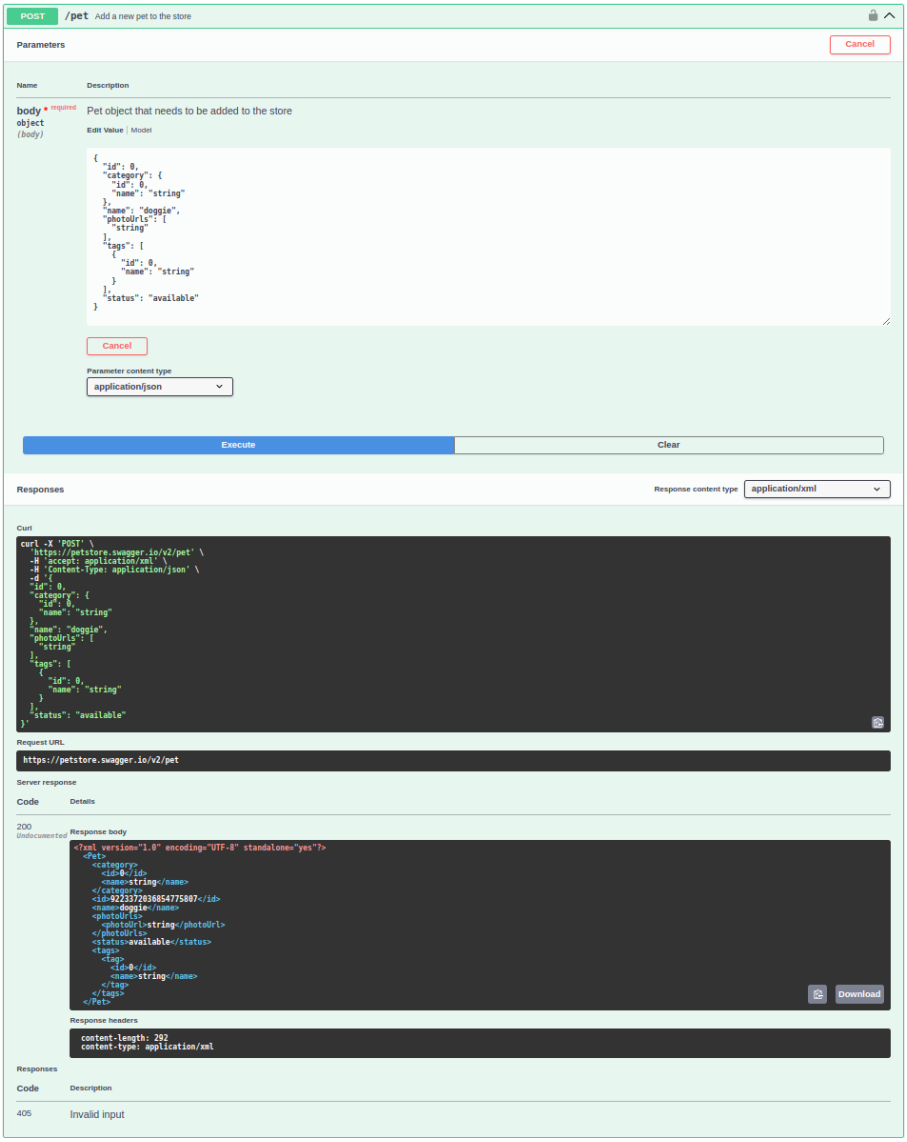
KUVA 8. Rajapinnan tiedot, metadata ja polut. Kuvassa on Swagger UI -rajapintadokumentaatio Swagger Petstore -nimisestä rajapinnasta. Rajapintadokumentaatiossa näkyy pet- ja store-resurssien päätepisteitä.

Poluista on myös mahdollista avata näkymä polun yksityiskohtaisista tiedoista. Tiedoista näkyy polun vastaanottamat kuvatut parametrit, joista on myös mahdollista näkyä pyyntörunkojen esimerkkiobjektit, ja päätepisteen vastauksien tiedot esimerkkeineen ja virhekoodeineen. (Kuva 9.)



KUVA 9. Rajapinnan polun yksityiskohtaiset tiedot Swagger UI:ssa. Kuvassa näkyy yksittäisen pet-resurssin uploadImage-päätepisteen dokumentaatio. Dokumentaatio päätepisteestä on jaettu kahteen osaan: parametrit, jota päätepiste ottaa vastaan kutsujen yhteydessä, ja kuvaus palvelimen palauttamasta vastauksesta kyseiseltä päätepisteeltä.

Polkukohtaisen tietonäkymän kautta on myös mahdollista tehdä pyyntöjä kyseisen polun takana olevaan päätepisteeseen. Swagger UI tekee pyynnön curl-komentona ja pyynnön palautusarvo näytetään käyttöliittymässä. (Kuva 10.)



KUVA 10. Pyynnön tekeminen päätepisteeseen ja päätepisteen vastaus pyyntöön Swagger UI:ssa. Pyyntö on tehty pet-resurssin juuripolkuun uuden objektin luomiseksi. Kuvassa näkyy JSON-objektissa, mitä dataa palvelimelle lähetetään, miten pyyntö lähetetään curl-työkalulla ja miten palvelin vastaa pyyntöön onnistuneesti 200-tilakoodilla ja luodun objektin tiedoilla.

Swagger UI myös tarjoaa osion järjestelmän tietomalleista, jotka julkaistaan rajapintojen kautta. Osio näyttää tietomallien nimet, kentät ja kenttien tyypit. Jos kentät ovat relaatioita toisiin tietomalleihin, näytetään nämä tietomallin sisällä sisäkkäisinä tietomallitietoina. (Kuva 11.)

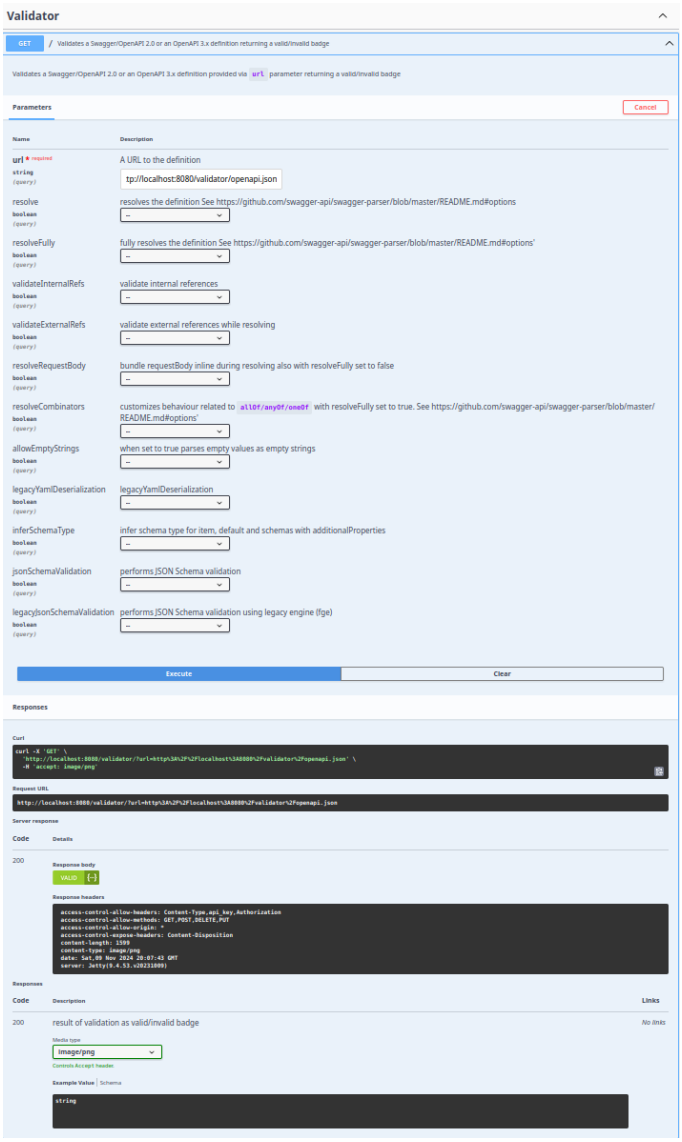
KUVA 11. Rajapinnan kautta julkaistujen tietomallien osio Swagger UI:ssa. Kuvassa näkyy tietomallit ApiResponse, Category, Pet, Tag, Order ja User. Jokaisesta tietomallista on näkyvissä myös tietomallin kentät ja kenttien tyypit.

Jotta dokumentaation käyttöliittymästä olisi mahdollisimman paljon hyötyä, on kuitenkin tärkeää, että kehitystiimit sopivat ja suunnittelevat yhdessä, että millä tavalla rajapinnat dokumentoidaan, jotta dokumentaatio ja sen käytettävyys vastaa niitä odotuksia, mitä kehitystiimin henkilöillä on rajapintadokumentaatiosta. Tämä koskee etenkin rajapinnan metadatoja, kuvauksia ja aihetunnisteita. (Rosenstock & Ponelat 2022.)

2.2.3. Muu OpenAPI-ekosysteemi ja -työkalut

Automaattisen dokumentaatiokäyttöliittymägeneroinnin lisäksi OpenAPI-spesifikaation noudattaminen kuvaamalla rajapinta OpenAPI-dokumenttiin tarjoaa paljon laajempaakin työkalujen ekosysteemiä. Näitä ovat esimerkiksi OpenAPI-dokumentaation validointi, rajapinnan läpi kulkevan datan validaatio, rajapinnoista luotavan SDK-kirjaston generointi, palvelimista tehtäviä jäljitelmäsimulaatioita ja staattisia analyysiauditointeja OpenAPI-dokumenttia vasten. (OpenAPI Intiative n.d.) Näitä työkaluja käydään lyhyesti läpi tässä luvussa.

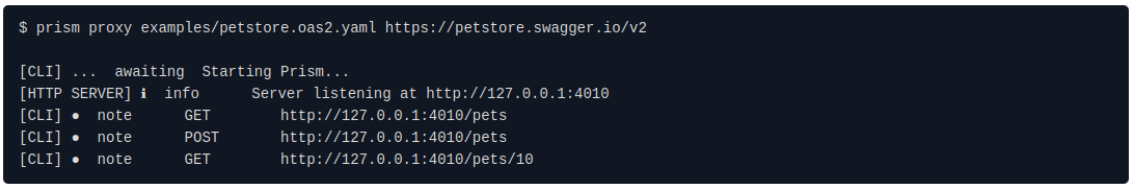
OpenAPI-dokumentti toimii niin kutsuttuna totuuden lähteenä ja sopimuksena järjestelmän kehittäjille rajapinnan toiminnoista, joten tähän dokumenttiin täytyy pystyä luottamaan. Tätä varten löytyy työkaluja, jotka parsivat niille annetun OpenAPI-dokumentin ja validoi sen virallista OpenAPI-spesifikaatioita vasten, jotta dokumentaation tekijä voi olla varma sen oikeellisuudesta. Validaation voi tehdä esimerkiksi osana kehittäjän lokaalia kehitysprosessia tai testinä osana CI/CD-putkea, jotta tuotantoon ei julkaista virheellistä OpenAPI-dokumentaatiota, jota muut kehittäjät käyttävät. Tästä työkalusta on esimerkkinä Swaggerin oma Swagger Validator Badge -työkalu, jota voi käyttää suoraan projektin verkkosivulta löytyvän käyttöliittymän kautta, curl-työkalun avulla tai tekemällä validaation omalla työasemalla Docker-kontin kautta. Validaation lisäksi työkalun kautta voi hankkia itselleen validaatiomerkin, jonka avulla validaatiotiedon saa visuaalisesti näkyville. (Kuva 12.)



KUVA 12. OpenAPI-dokumentin validointi Swagger Validator Badge -työkalulla. Työkalussa näkyy saman tyyliset osiot kuin Swagger UI -työkalussa, mutta vain konfiguroituna OpenAPI-dokumentin validointitietoja varten. Työkalun vastauksena palautetaan merkki dokumentin validaatiotilasta.

Itse OpenAPI-dokumentin validoinnin lisäksi on myös olemassa työkaluja, jotka validoivat OpenAPI-dokumenttia vasten dataa sekä palvelimelle tulevista pyynnöistä että palvelimelta takaisin asiakkaalle lähtevistä vastauksista kuvattua. Molemmat näistä validoinneista ovat tärkeitä. Järjestelmään tulevaan syötteeseen ei voi koskaan luottaa ja rajapinnan käyttäjän tulee voida luottaa siihen, että rajapinta palauttaa juuri sen datan, minkä se on luvannut palauttaa. (Rosenstock & Ponelat 2022.) Datavalidointiin löytyy useita eri tapoja. Prism on kokoelma useita eri OpenAPI-työkaluja, joista yksi on OpenAPI-validaatioon erikoistunut välityspalvelin. Välityspalvelin ottaa vastaan sekä asiakkaan lähettämät pyynnöt että

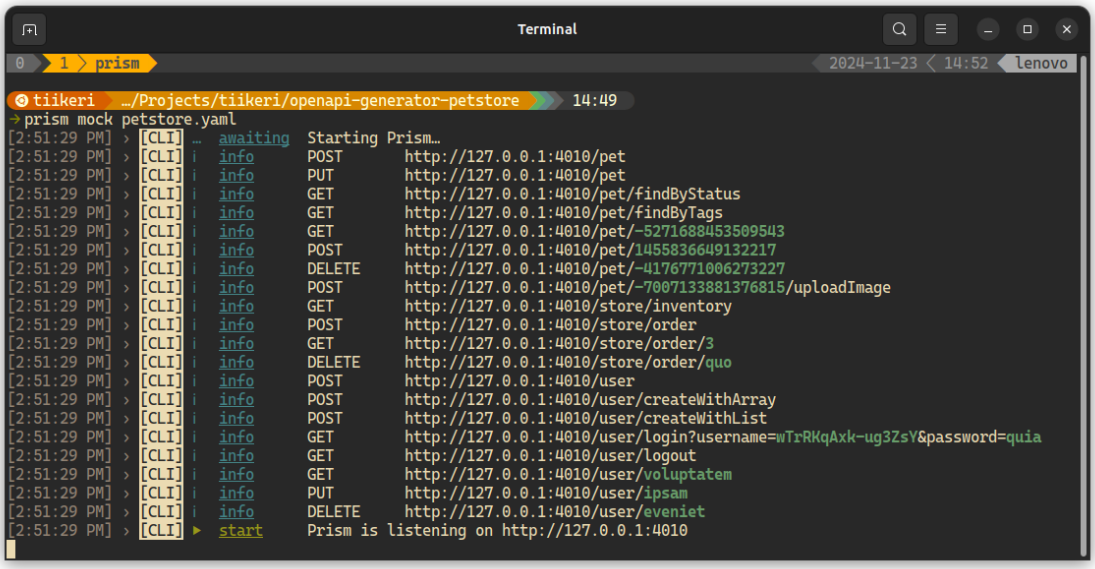
palvelimen vastaukset ja validoi molemmat annettua OpenAPI-dokumenttia vasten. Välityspalvelin ei kuitenkaan nosta validointivirheistä virheitä tai poikkeuksia vaan validaatiovirheet lokitetaan, joihin voi monitoroinnin kautta reagoida. Prism-työkalun tarjoama OpenAPI-validaatioon tarkoitetut välityspalvelimet eivät siis häiritse palvelun toimintaa vaan enemmänkin ilmoittavat lokien kautta, jos asiakkaan lähettämä pyyntö tai palvelimen lähettämä kutsu ja täten palvelimen rajapintaimplementaatio ei vastaa odotettua rajapintaspesifikaatiota. (Prism n.d.) Häiritsemättömän välityspalvelimen ohella on myös vaihtoehtoja, jotka toimivat eri toimintaperiaatteella. Esimerkiksi OpenAPI-core (n.d.) tarjoaa Python-sovellusten kehittäjille mahdollisuuden päättää itse, miten haluavat sovelluksensa hoitaa virheet, jotka johtuvat validaatiovirheistä, ja käyttää ohjelmallista validointia sovelluksen testisetissä. PayU GPO:n (n.d.) openapi-validator-middleware-paketti puolestaan toimii Node-pohjaisten webkehyksien, Expressin, Koan ja Fastifyn, kanssa väliohjelmistona, joka käy läpi sille tulleet pyynnöt ja vastaukset, ja validointivirheen sattuessa validointivirhe välitetään virheenkäsittelijälle, joka palauttaa asiakkaalle sopivan vastauksen. Tämä on tehokas tapa hoitaa validaatiovirheet, sillä väliohjelmiston kautta sovellus estää aktiivisesti kaikki pyynnöt ja vastaukset, jotka eivät ole rajapintaspesifikaation vaatimusten mukaisia. (Prism n.d.)



KUVA 13. Esimerkkikuva Prism-välityspalvelimen käytöstä. Kuvassa Prism-välityspalvelin käynnistetään komentoriviltä prism proxy -komennolla. Komento tulostaa tietoja välityspalvelimen Petstore-rajapinnasta osoitteeseen http://127.0.0.1:4010 ja sen päätepisteistä, joita voi käyttää kehityksessä. Tulosteessa näkyy myös pets-resurssin juuripolkuun osoitetut GET- ja POST-päätepisteet sekä yksittäisen pets-resurssin GET-päätepiste. (Prism n.d.)

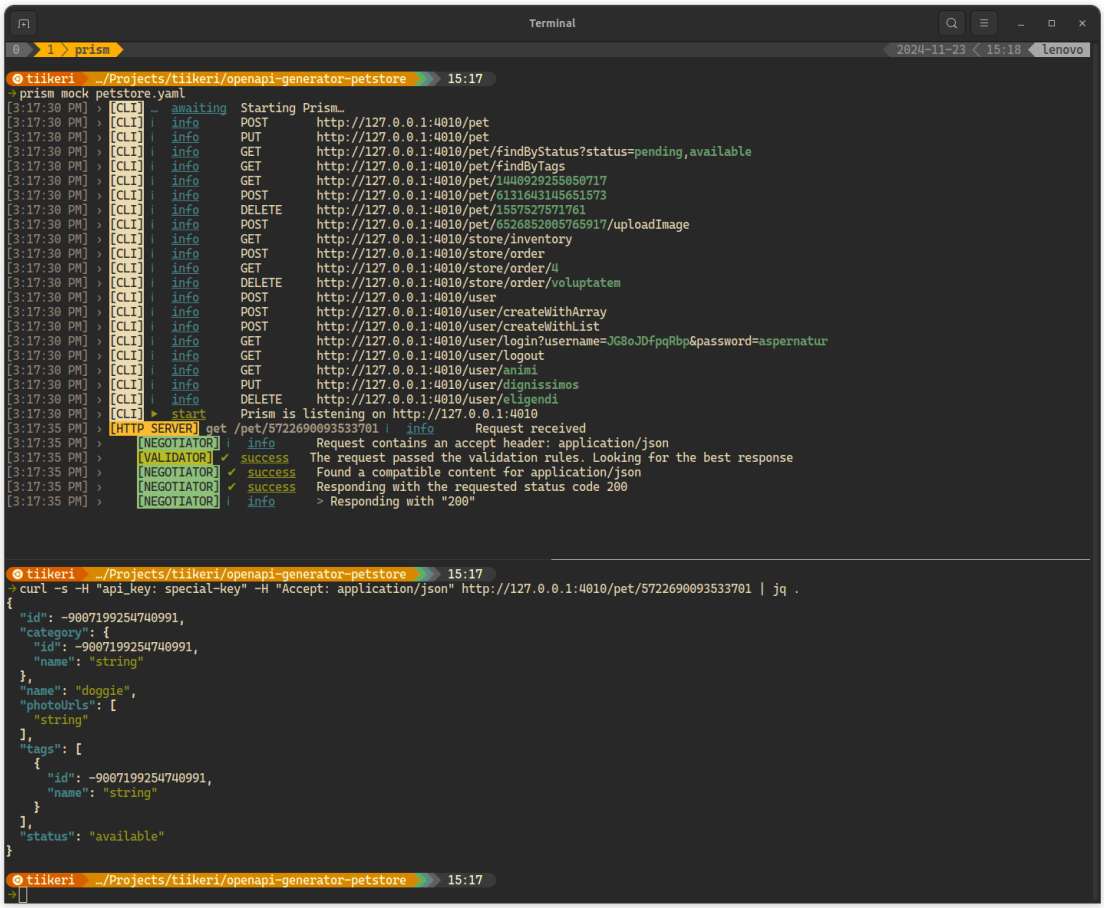
Validointiin tarkoitetun välityspalvelimen lisäksi Prism-työkalukokoelma tarjoaa myös jäljittelypalvelimen, joka simuloi HTTP-palvelimella pyörivää rajapintasovellusta. OpenAPI-dokumentin kuvausten perusteella jäljittelypalvelin tekee HTTP-

pyyntöjen validointia ja luo HTTP-vastauksia takaisin palvelimen rajapintakäyttäjälle. (Rosenstock & Ponelat 2022.)



KUVA 14. Prism-jäljittelypalvelimen käynnistäminen. Kuvassa jäljittelypalvelin käynnistetään prism mock -komennolla Petstore-rajapinnasta. Tulosteessa näkyy kaikki resurssit, mitä jäljittelypalvelin tarjoaa sen asiakkaalle. Resursseja on kolme kappaletta, pet, store ja user, ja jokaisella resurssilla on useita eri päätepisteitä.

Kuvassa 15 Prism-jäljittelypalvelinta vasten tehdään esimerkkinä HTTP-pyyntö käyttäen cURL-työkalua. Jäljittelypalvelin tunnistaa ja vastaanottaa palvelimelle tulevan kutsun, tunnistaa, että käyttäjä haluaa palvelimen vastaavan takaisin JSON-formaattina, tekee pyynnölle auktorisointi- ja parametrivalidoinnin ja palauttaa takaisin vastauksen haluttuna JSON-formaattina.



KUVA 15. HTTP-pyynnön tekeminen Prism-jäljittelypalvelimelle ja jäljittelypalvelimen HTTP-vastauksen luonti JSON-formaattina. Kuvassa näkyy GET-kutsun tekeminen curl-komennolla jäljittelypalvelinta vasten yksittäisestä pet-resurssista. Palvelin vastaa kutsuun palauttamalla pyydetyn resurssin ja jäljittelypalvelin omassa tulosteessa näkyy kutsun käsittelyyn liittyvää lokitusta.

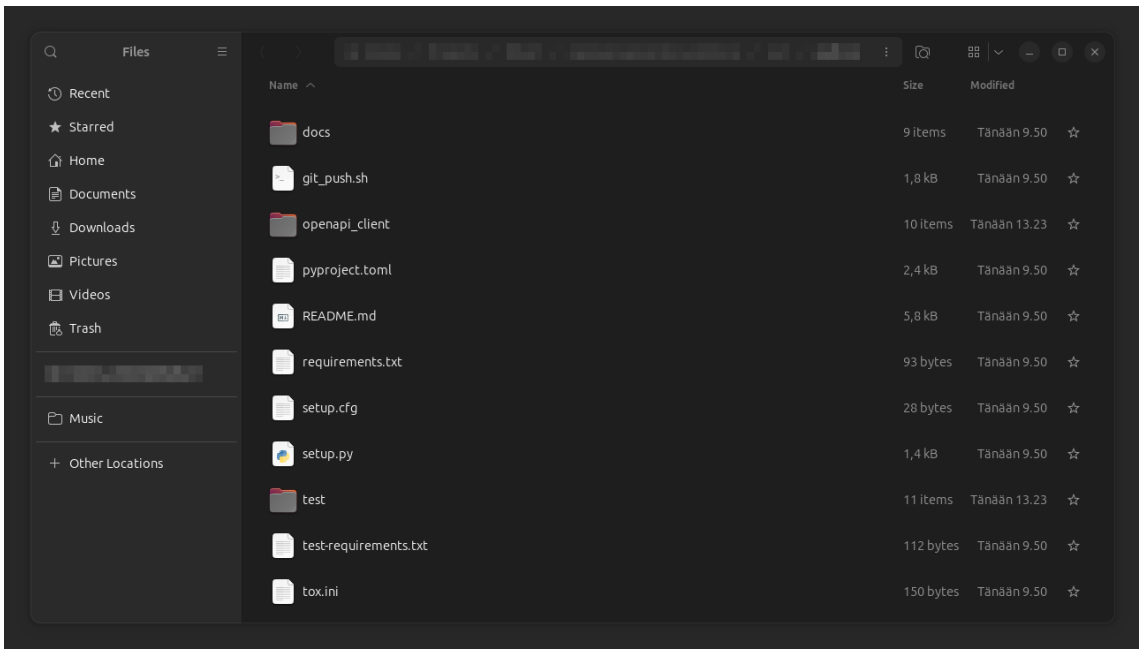
Normaalisti Prism-jäljittelypalvelimet palauttavat käyttäjälle takaisin dataa, joka on staattista käyttämällä OpenAPI-kuvausten HTTP-vastausten ja tietomallien esimerkkiarvoja. Palautettavia arvoja voi kuitenkin konfiguroida myös dynaamisiksi ajamalla Prism-jäljittelypalvelinta dynaamisessa moodissa ja ohjaamalla jäljittelypalvelinta palauttamaan oikeanlaisia dynaamisia arvoja käyttämällä Faker- ja JSON Schema Faker -kirjastoja. (Prisma n.d.)

Jäljittelypalvelimien käyttö mahdollistaa järjestelmän nopean kehittämistahdin, sillä rajapintaimplementaation ei tarvitse olla edes valmis, kun esimerkiksi käyttöliittymäpuolen implementoija voi jo aloittaa käyttöliittymäpuolen kirjoittamisen

palvelinta vasten siten, että kutsut käyttöliittymästä palvelimelle palauttavat oikeita arvoja. Käyttöliittymä- ja taustajärjestelmä voi täten kehittää siis samanaikaisesti. (Rosenstock & Ponelat 2022.)

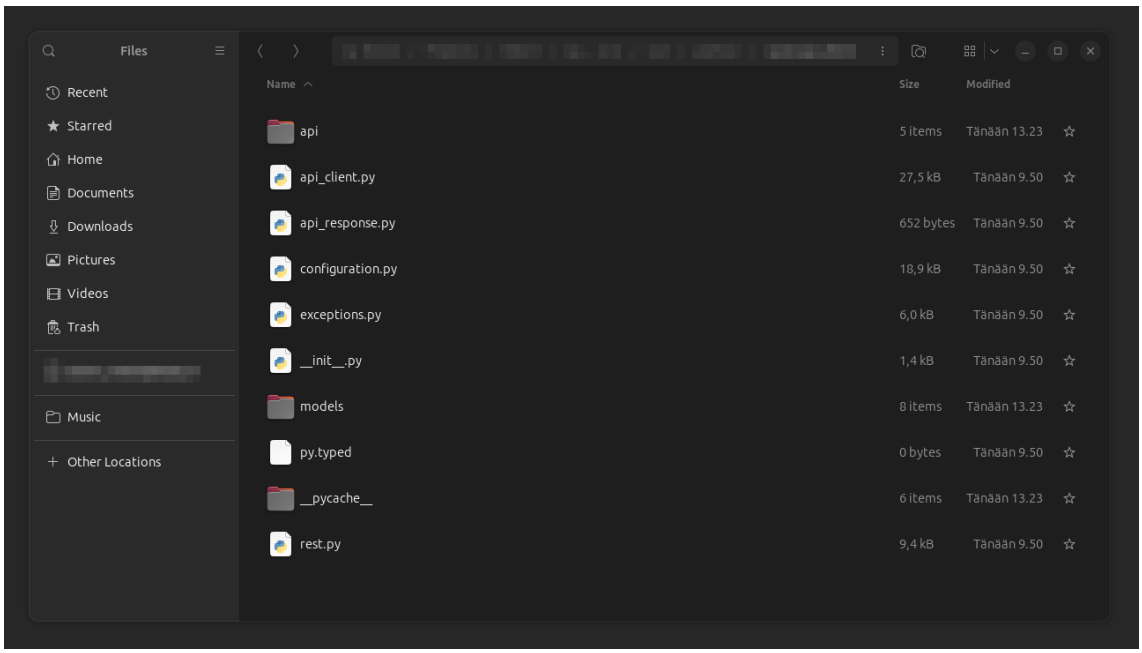
Rajapintojen käyttäjät joutuvat usein käyttämään rajapintaa siten, että he tekevät rajapintaan suoria HTTP-kutsuja niillä työkaluilla tai ohjelmointikielillä, mitä on mahdollista käyttää. Näitä tapoja on esimerkiksi JavaScriptin Fetch-rajapinta tai cURL-työkalulla. Rajapintakutsua tehtäessä näillä tavoilla kehittäjä joutuu itse huolehtimaan monista eri asioista kutsua ajatellen: pyynnön URL-polku, HTTP-otsakkeet, serialisointi, autentikointi ja auktorisointi, manuaalinen virheiden tarkastus HTTP-vastauksesta, ja manuaalinen deserialisointi HTTP-vastauksesta. Tämän kaiken pystyy abstraktoimaan, jotta rajapintakäyttäjän ei tarvitse kaikkea tätä hoitaa itse. Tämä abstraktio voidaan toteuttaa SDK-työkaluilla. SDK:t ovat kokoelma ohjelmistokehityskirjastoja, työkaluja ja dokumentaatiota, jotka helpottavat ja nopeuttavat kehitystyötä ja vähentävät järjestelmien virheherkkyyttä. SDK-kirjastoja toteutetaan eri ohjelmointikielillä, jotta mahdollisimman moni ohjelmistokehittäjä pystyy hyödyntämään valmiita ohjelmistokirjastoja. SDK-kirjastot voivat toteuttaa ohjelmointikielikohtaisen abstraktion erityyppisistä rajapinnoista, esimerkiksi käyttöjärjestelmä-, alusta- ja REST-rajapinnoista. OpenAPI-spesifikaation mukaisten rajapintojen kohdalla on kyse REST-rajapinnoista. OpenAPI-ekosysteemistä löytyy työkaluja, joiden avulla on mahdollista luoda OpenAPI-spesifikaation mukaisesti kuvatusta REST-rajapinnasta SDK-kirjaston, jonka avulla rajapinnan käyttäjä voi itselleen tunnetulla ohjelmointikielellä kutsua SDK-kirjaston metodeja, jotka toteuttavat sisäisesti HTTP-kutsut sisältäen suurimman osan niistä asioista, mitä kehittäjä joutuisi itse manuaalisti tekemään. (Rosenstock & Ponelat 2022.)

OpenAPI Generator on yksi esimerkki avoimen lähdekoodin työkalu, jolla voi määritellä useista eri ohjelmointikielivaihtoehdoista omansa ja luoda valitulle kielelle SDK-kirjasto (OpenAPI Generator 2024).



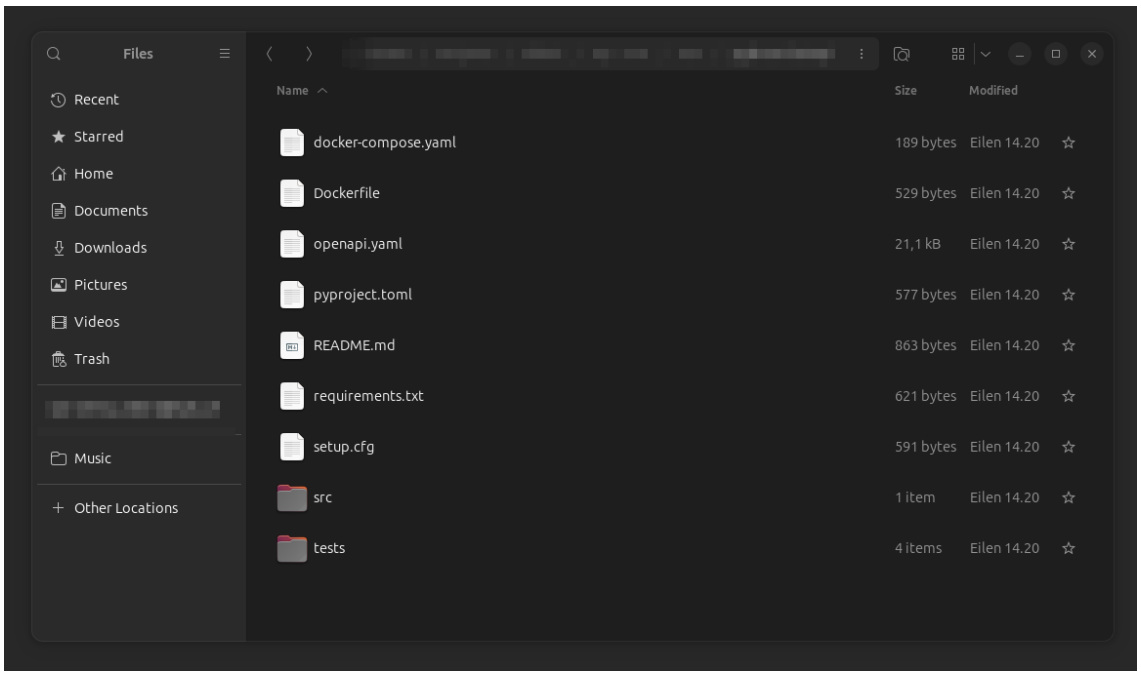
KUVA 16. OpenAPI Generatorin luoma SDK-kirjaston tiedostorakenne. Tiedostorakenne on SDK-kirjaston tiedostopolun juuresta, joka sisältää ympäristöön ja riippuvuuksiin liittyviä tiedostoja, konfiguraatiotiedostoja sekä kehitykseen. Tämän lisäksi tiedostorakenne sisältää dokumentaatiohakemiston sekä luodun SDK-kirjaston ydinasian eli rajapinnan kanssa keskusteluun tarkoitetun räätälöidyn lähdekoodin oman openapi_client-hakemiston alla.

Olennaisimmat asiat luodusta SDK:sta on rajapinnan ja SDK-kirjastototeutuksen dokumentaatio, kirjaston käyttämät kolmannen osapuolen paketit, ja SDK-kirjaston pääasiallinen moduuli, joka sisältää keskitetyn rajapintakutsujen mahdollistavan ApiClient-luokan, HTTP-kutsuja käsittelevät HTTP-verbikohtaiset metodit, rajapintojen palauttamat deserialisointiluokat, kirjaston ja REST-rajapinnan käyttämät konfiguraatiot, virheluokat ja tietomallit, jotka rajapintaan OpenAPI-dokumenttiin on määritelty, ja joita käytetään deserialisointiluokkina HTTP-pyyntöjen palautettavissa vastauksissa (kuva 17).



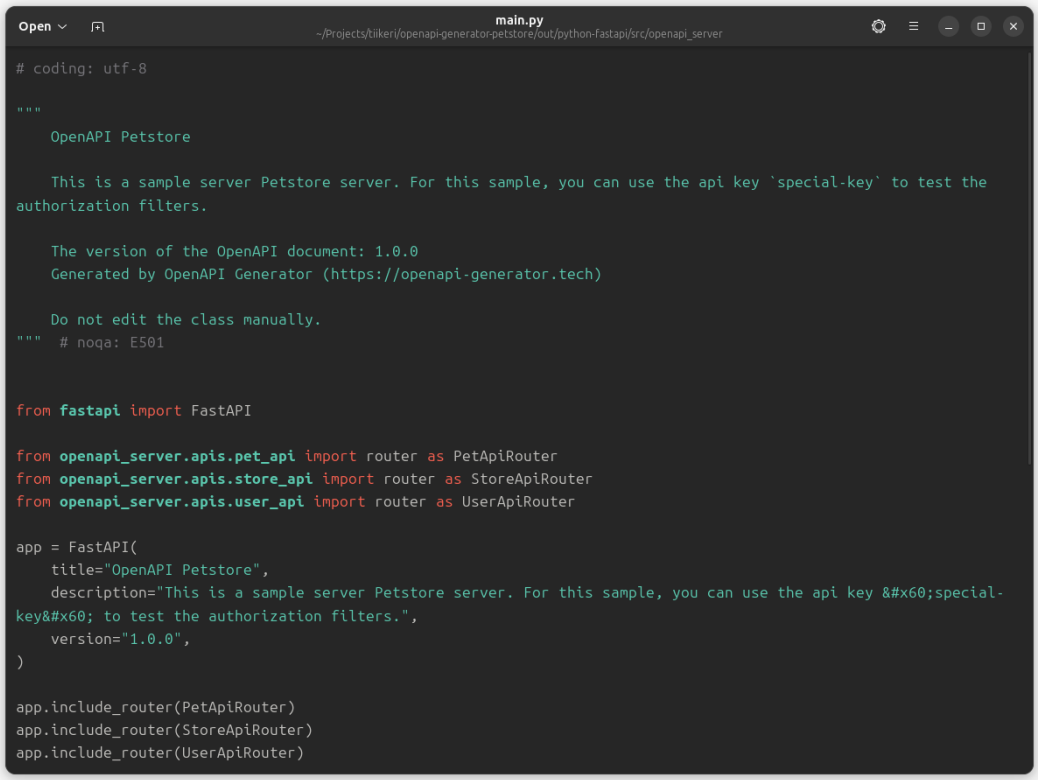
KUVA 17. OpenAPI Generatorin luoman SDK-kirjaston openapi_client-moduuli, joka toteuttaa osuuden rajapinnan. Hakemisto sisältää konfiguraatiotiedostoja, sovelluslogiikkaa sekä rajapinnan tarjoamien resurssien tietomalliluokat omassa models-hakemistossaan.

Rajapintakäyttäjiä hyödyttävien SDK-kirjastojen lisäksi myös REST-rajapintaa implementoivat kehittäjät pystyvät hyödyntämään REST-rajapintojen OpenAPI-kuvauksia nopeuttamaan rajapintatoteutuksia. Tähän käyttötapaukseen on luotu työkaluja, jotka pystyvät SDK-kirjastojen luonnin tavoin luomaan niin kutsuttuja palvelinrunkoja eli pohjatoteutuksia rajapinnoille. Esimerkiksi OpenAPI Generator pystyy SDK-kirjastojen lisäksi luomaan myös palvelinrunkoja. OpenAPI Generatorin luomat palvelinrungot sisältävät OpenAPI-kuvauksen pohjalta rajapinnan aloituspisteen, lisää siihen alamoduulit, jotka toteuttavat tiettyyn tietomalliin pohjautuvan resurssin polut ja käsittelijät, sekä muut tarvittavat osat rajapinnan toimintaa varten, kuten tietomallit ja yleinen luokka rajapintavastauksille, rajapinnan pohjatoteutukseen vaaditut kolmannen osapuolen kirjastot sekä yksikkötestit. (kuva 18.)



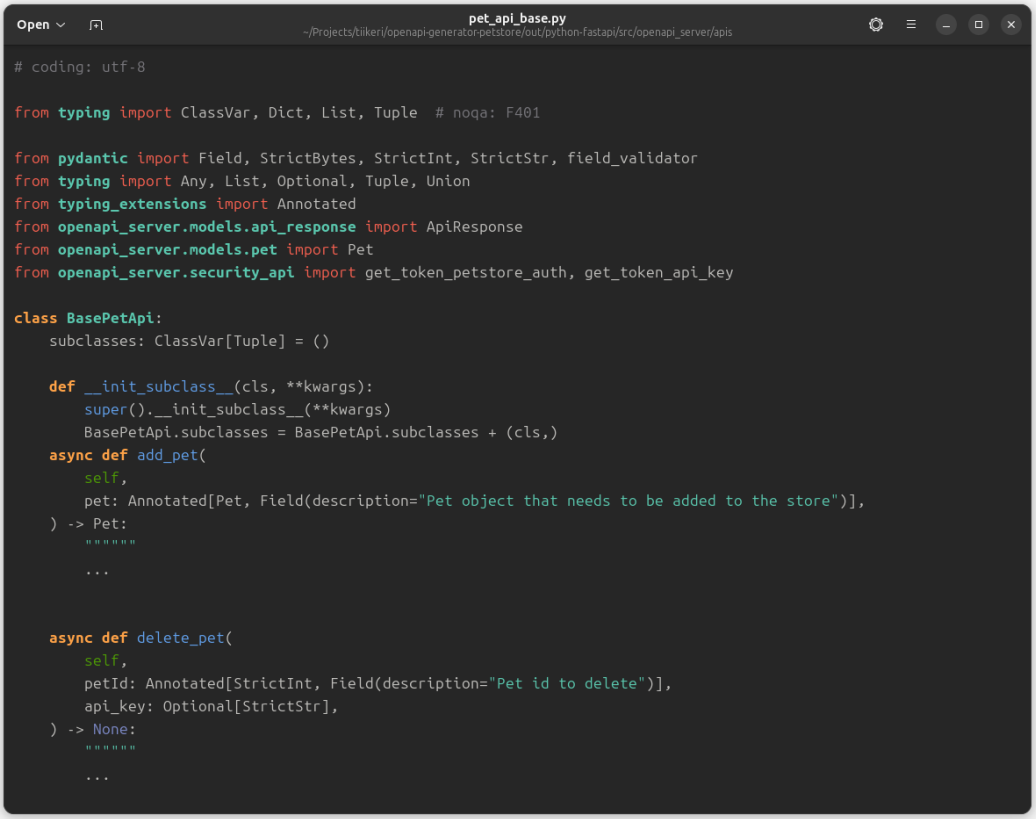
KUVA 18. OpenAPI Generatorin luoman palvelinrungon tiedostorakenne. Palvelinrungon juuressa on konfiguraatiotiedostoja sekä pääasiallisen lähdekoodin src-hakemisto että palvelinrunkolähdekoodin testien tests-hakemisto.

Esimerkiksi Python-pohjaisen FastAPI-palvelinrungon aloituspistemoduulissa on käytetty suoraan demokäyttöön tarkoitetun OpenAPI-dokumentin kentttiä niin lähdekoodin omassa dokumentaatiossa kommenttien avulla sekä rajapintasovelluksen instanssissa, jonka attribuuteille annetaan kuvaustekstejä. Aloituspisteeseen on lisätty myös Pet-, Store- ja User-tietomalleihin pohjautuvat polut ja käsittelijät, jotka ovat myös toteuttu palvelinrunkoon automaattisesti. (kuva 19).



KUVA 19. OpenAPI Generatorin luoman palvelinrungon aloituspistemoduuli. Moduulissa on alustettuna FastAPI-rajapintaluokka, johon on sisällytetty kaikkien rajapinnan tarjoamien resurssien päätepisteiden reitittimet. Tiedosto sisältää myös dokumentaatiota.

Kyseessä on kuitenkin vain palvelinrunko. Työkalu, joka luo pohjatoteutuksen rajapinnalle, ei kuitenkaan tiedä pelkän OpenAPI-dokumentin perusteella, miten esimerkiksi resurssien käsittely tulisi hoitaa HTTP-kutsujen käsittelymetodeissa. (Rosenstock & Ponelat 2022.) Tämän takia palvelinrunko luo nimensä mukaisesti vain rajapinnan toteuttavien luokkien rungot, jotka kehittäjän täytyy täydentää halutulla käsittelylogiikalla. (kuva 20.)



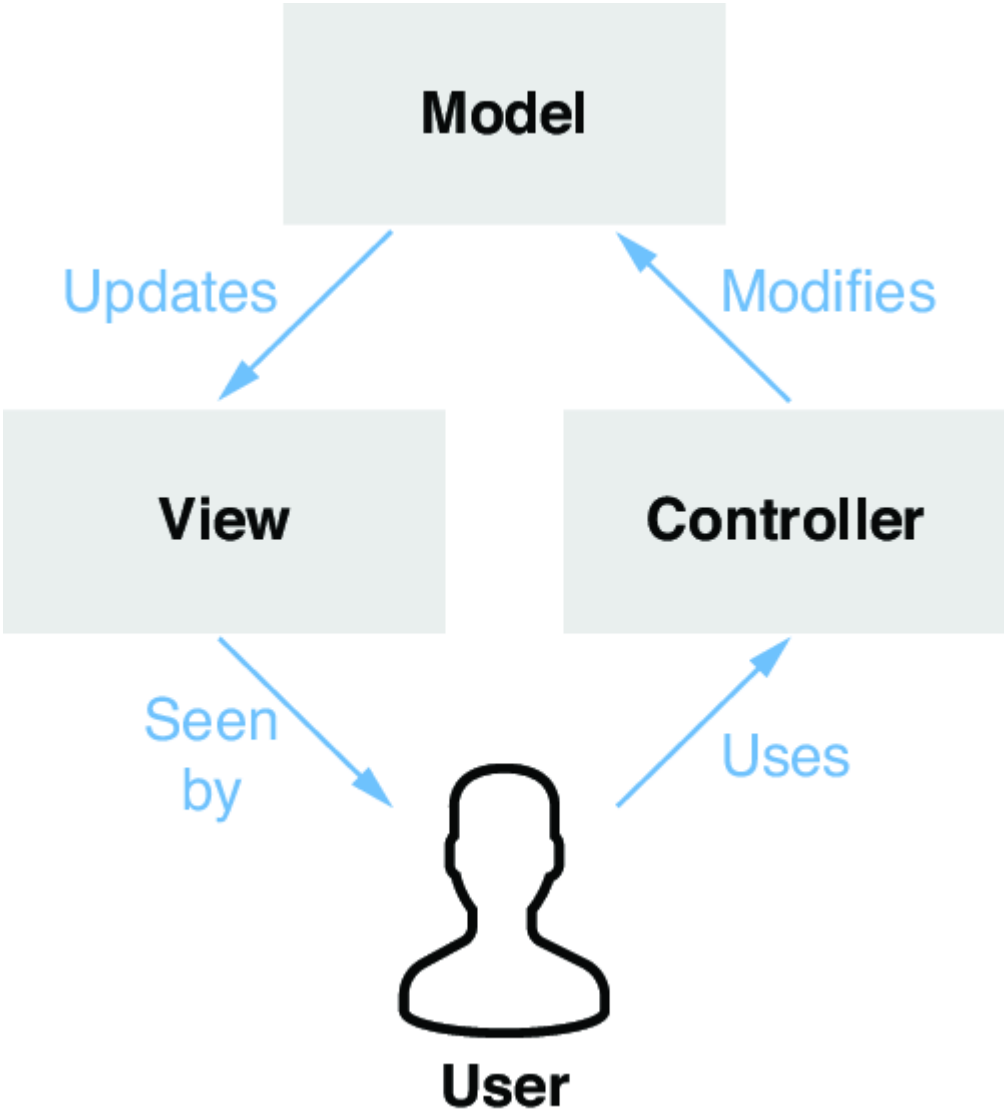
KUVA 20. OpenAPI Generatorin luoman palvelinrungon kutsujen käsittelijöiden rungot. Kuvassa esitellään pet-resursseihin liittyvä luokka, johon on määritelty käsittelijämetodit. Kuvassa näkyy esimerkkiluokan delete_post-metodi, jonka tarkoituksena on määritellä rajapinnan pet-polkuun tehtävän DELETE-pyynnön logiikka. Rungossa ei ole ennaltakirjoitettua koodia.

Tämä kuitenkin nopeuttaa toteutusta huomattavasti, kun kehittäjien ei tarvitse käyttää aikaa luokkien ja metodien nimeämiseen ja rakennemäärittelyyn. SDK:n ja palvelinrungon avulla järjestelmän molemmat puolet, niin taustajärjestelmän rajapintaimplementointi ja tietomallit kuin käyttöliittymän rajapintakutsujen ja -vastausten käsittely ja deserialisointi käytettäviksi tietomalleja vastaaviksi objekteiksi, saadaan hyvin nopeasti alkuun OpenAPI-dokumentin avulla. Useimmiten sovelluskehitysprojekteissa juuri tämänlaiset toteutukset toistuvat projektien toiseen, joten tälla tavalla lähdekoodin automaattinen luominen vapauttaa projekteissa budjettia ja aikaa niihin asioihin, mitkä oikeasti tekevät sovelluksesta uniikin.

2.3. Django ja Django Rest Framework

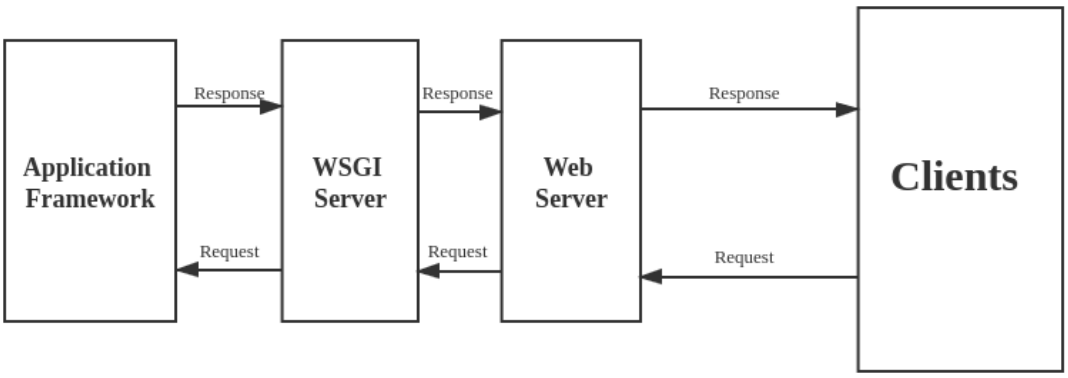
Tässä kappaleessa käsitellään Django-verkkokehystä ja siihen olennaisesti liittyvää Django Rest Framework -kirjastoa. Nämä teknologiat on valittu tähän opinnäytetyöhon sen takia, että opinnäytetyön toimeksiantaja, Haltu Oy, on tuottanut hyvin paljon ohjelmistoja juuri Django-kehyksellä. Seuraavassa pääkappaleessa kuvataan käytännön tasolla, millä tavalla Haltulla mahdollistetaan OpenAPI-spesifikaation mukaisten rajapintojen kuvaus sekä hallinta järjestelmissä, jotka pohjautuvat Django-kehykseen. Tämän kappaleen tarkoituksena on taustoittaa seuraavaa pääkappaletta.

Django on Python-ohjelmointikielellä kirjoitettu ohjelmistokehys verkkosovellusten kehittämiseen. Siinä, missä ohjelmistokirjastot tuovat lisää työkaluja kehittäjälle osaksi lähdekoodia, ohjelmistokehyksillä on useimmiten valmis rakenne, jota kehittäjän täytyy noudattaa. Rakenne toimii abstraktiona sille, miten ohjelmistokehys toteuttaa sen tarkoitusta, jonka tavoitteena on helpottaa ja nopeuttaa kehitystyötä. Verkkosovelluskehysten kohdalla ohjelmistokehykset yleensä tarjoavat tavan hoitaa tyypilliset toiminnallisuudet, kuten HTTP-kutsujen ja -vastausten käsittely, tietokantaskeemat ja -yhteydet, ja käyttöliittymänäkymien rakentamisen. Tyypillinen arkkitehtuurimalli, jota usein käytetään ohjelmistojen suunnittelussa ja rakentamisessa, onkin MVC-arkkitehtuurimalli. MVC tulee sanoista model-view-controller eli malli-näkymä-käsittelijä. Malli kuvastaa sitä, miten järjestelmä kuvaa ja tallentaa tarjoamaansa tietoa, etenkin tietokantatasolla. Näkymä esittää järjestelmään tallennettua tietoa käyttäjälle halutussa muodossa ja tarjoaa tavan vastaanottaa tietoa käyttäjältä. Käsittelijä puolestaan ottaa käyttäjän syötteet vastaan ja muuttaa järjestelmään tallennettua tietoa ja tarjoaa näkymän käyttäjälle. (Trudeau 2024). Djangon toimintalogiikka on hyvin lähellä MVC-arkkitehtuuria, vaikka Djangon tulkinnan mukaan MVC-arkkitehtuurimalli ei sovi suoraan Djangoon (Django n.a.).



KUVA 21. MVC-arkkitehtuurikaavio. Kaavio kuvaa MVC-arkkitehtuurin seuraavasti: käyttäjä haluaa muokata järjestelmän tietoa, mutta käyttäjä ei tee muutosta suoraan, vaan käyttäjä käyttää kontrolleria tietokannan rajapintana. Kontrolleri tekee lopullisen muutoksia tietokantaan, joka puolestaan vaikuttaa siihen tietoon, mitä käyttäjälle näytetään. (Trudeau 2024.)

Verkkosovelluksen, joka on toteutettu Djangolla, järjestelmäarkkitehtuuri näyttää korkealta tasolta kuvattuna kuvaan 22 (Zhou 2021) kuvatulta prosessilta:

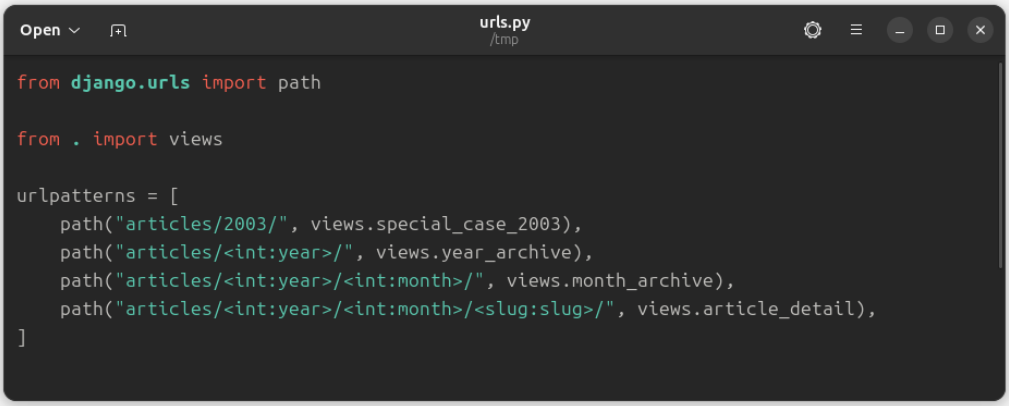


KUVA 22. Prosessikuva HTTP-pyynnön näkökulmasta perinteisessä Django-järjestelmässä arkkitehtuuritasolla. Asiakkaan HTTP-pyyntö menee ensiksi webpalvelimelle, joka ohjaa pyynnön WSGI-palvelimelle, jonka tehtävä on muuntaa HTTP-pyyntö Python-luokaksi, jotta Django-sovellus voi käsitellä pyynnön. Django käsittelee HTTP-pyynnön ja lähettää HTTP-vastauksen takaisin WSGI-palvelimelle, jotta sen voi muuttaa takaisin muotoon, jonka voi lähettää takaisin asiakkalle webpalvelimen kautta. (Zhou 2021.)

Käyttäjän tehdessä minkä tahansa HTTP-pyynnön sovellukselle, pyyntö kulkee normaalisti HTTP-palvelimelle, jonka kautta pyyntö ohjataan WSGI-palvelimelle. WSGI on lyhenne sanoista Web Server Gateway Interface, joka tarkoittaa verkkopalvelimen yhdyskäytävärajapintaa. WSGI-palvelimen tehtävänä on nimensä mukaisesti toimia siltana tai tulkkina HTTP-palvelimen ja Djangon välillä. WSGI parsii HTTP-palvelimelta annetusta HTTP-pyynnöstä tiedot, rakentaa tiedoista Python-tulkin ymmärtävä tietorakenne ja kutsuu WSGI-yhteensopivaa rajapintaa Django-sovelluksesta. Tämä välivaihe on pakollinen, sillä Django ei itse ymmärrä raakaa HTTP-pyyntöä. HTTP-pyyntö muutetaan Djangossa HttpRequest-luokaksi, ja pyyntö käsitellään Djangossa, jossa muodostetaan myös HTTP-vastaus HttpResponse-luokkana, ja tämä vastaus lähetetään takaisin WSGI-palvelimelle, joka muodostaa Python-luokasta oikean HTTP-vastauksen, jonka HTTP-palvelimen lähettää takaisin vastauksena käyttäjälle. (Trudeau 2024.)

Django käsittelee HTTP-pyynnöt siten, että ensiksi Django pyrkii löytämään pyynnön URL-polulle vastaavan pyynnönkäsittelijän. Django pyynnönkäsittelijät konfiguroidaan vastaamaan jotain tiettyä polkua. Näitä kutsutaan URL-konfiguraatioiksi, jotka useimmiten määritellään urls.py-nimisissä tiedostoissa. Django siis pyrkii löytämään sovelluksen URL-konfiguraatioista polun, joka vastaa HTTP-

pyynnön polkua, ja kutsuu pyynnönkäsittelijäfunktiota antamalla sille HTTP-pyyntöä vastaavan Python-luokan funktion argumentiksi, jota funktio alkaa käsittelemään. URL-konfiguraatiot pystyvät myös tunnistamaan dynaamisesti polulle annettuja määreitä, jotka annetaan lisäargumentteina pyynnönkäsittelijöille. (Trudeau 2024.) Esimerkiksi kuvassa 23 HTTP-pyynnön polusta /articles/2024/ view_archive-pyynnönkäsittelijäfunktiolle annettaisiin nimetty year-argumentti, jota pyynnönkäsittelyssä voidaan suoraan käyttää ilman erillistä polun parsimista.

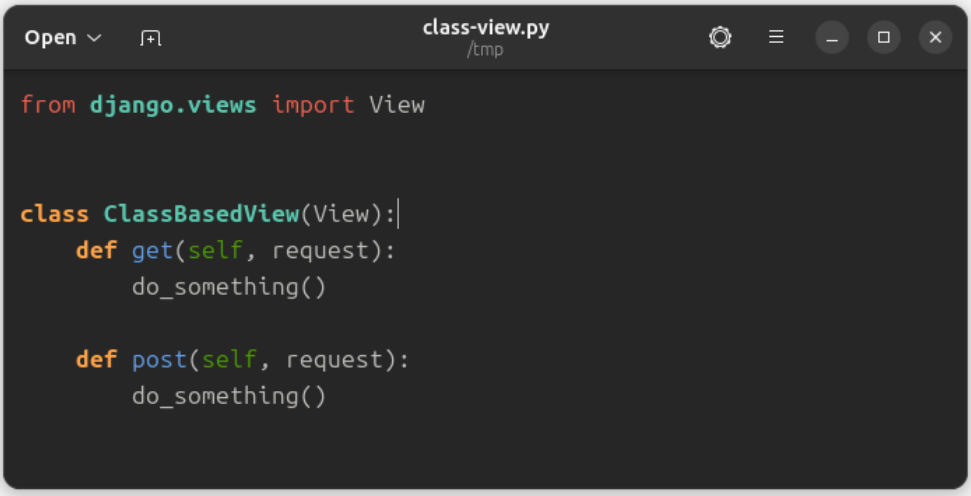


KUVA 23. Esimerkki Djangon URL-konfiguraatioista. Kuvassa näkyy lista neljästä määritellystä polusta artikkeliresursseja varten. Määrittellyistä poluista löytyy yksi normaali, staattinen polku ja kolme parametrisoitua polkua, jotka ottavat vastaan eri parametreja asiakkailta, jotka käsitellään kutsun käsittelyn yhteydessä. (Django n.a.)

Käsittelijäfunktion tehtävänä on käsitellä HTTP-pyyntö sovelluksen toimintalogiikan mukaisesti, ja luoda vastaus, joka lähetetään takaisin käyttäjälle (Trudeau 2024). Pyynnönkäsittelijä voi olla joko funktio tai pyynnönkäsittelijästä voi tehdä myös luokkapohjaisen, jolloin luokalle toteutetaan ne HTTP-metodeja vastaavat metodit, kuten get, post ja niin edelleen, jotka käsittelevät vain kyseisen HTTP-metodin kutsuja. Jos HTTP-pyyntöjen käsittely tehdään funktiopohjaisesti, tulee käsittelijäfunktiossa manuaalisti käsitellä eri HTTP-metodien käsittely, kuten kuvassa 23 (n.d.) kuvataan.

KUVA 24. Esimerkkikoodia, miten funktiopohjaisessa pyynnönkäsittelijässä voidaan huomioida eri HTTP-metodit. Koodissa funktiossa määritellään kaksi mahdollista polkua, jota pitkin koodia suoritetaan. Suoritus perustuu siihen, onko pyynnön HTTP-metodi GET vai POST. (Django n.d.)

Kuvan 24 (n.d.) funktiopohjaista pyynnönkäsittelijää vastaava luokkapohjainen toteutus näyttää kuvan 24 mukaiselta toteutukselta.



KUVA 25. Esimerkkitoteutus luokkapohjaisesta pyynnönkäsittelystä. Esimerkissä on tiedosto, jossa on luokkapohjainen toteutus GET- ja POST-pyyntöjen käsittelijöistä. Käsittelijät ovat määriteltyinä itsenäisinä get- ja post-nimisinä funktioina, joita kutsutaan luokkatoteutuksen sisäisen logiikan mukaisesti, kun Django vastaanottaa kutsun asiakkaalta.

Kuvassa 26 (n.d.) pyynnönkäsittelijän lähettää käyttäjälle takaisin HTML-vastauksen, jossa kerrotaan, paljon kello on. Django lähettää HTTP-vastaukset HttpResponse-luokkaa käyttäessä oletuksena HTML-dokumentteina, sillä vastauksen Content-Type-otsakkeeksi tulee oletuksena text/html. Myös tilakoodi asetetaan oletuksena 200:ksi, kun sitä ei erikseen aseteta.

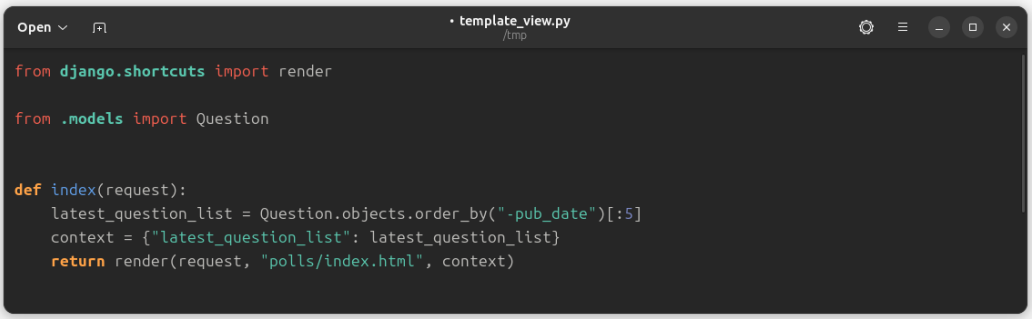
KUVA 26. Esimerkki Djangon pyynnönkäsittelijäfunktiosta, joka palauttaa käyttäjälle HTML-formaatissa vastauksen siitä, kuinka paljon kello on. Käsittelijä käyttää Pythonin standardikirjastoa selvittämään nykyisen ajan. Tämän jälkeen käsittelijä rakentaa HTML-merkkijonon nykyisellä ajalla, ja HTML-sisältö palautetaan takaisin käyttäjälle. (Django n.d.)

Djangon kohdalla onkin useimmiten kyseessä HTML-dokumentti, joka lähetetään takaisin vastauksena käyttäjälle. Joko kyseessä on suora GET-pyyntö halutusta sivusta HTML-dokumenttina tai sitten POST-pyyntö, jossa käyttäjä haluaa muokata tai lisätä uutta dataa järjestelmässä, jonka lopputuloksena myös käyttöliittymä päivitetään ja lähetetään käyttäjälle uudestaan. HTML-dokumentit ovat useimmiten pitkiä ja sivuilla on usein myös toistuvaa sisältöä, joten koko HTML-rakenteen kirjoittaminen pyynnönkäsittelijäfunktioon voi olla tehotonta. Tätä varten Django käyttää templaattimoottoria, joka pohjautuu Jinja2-templaattimoottoriin. Templaattien avulla kehittäjä voi luoda uudelleenkäytettäviä HTML-pohjia ja -tiedostoja sivujen käyttöliittymän rakentamiseen. Templaatit ottavat vastaan niin kutsutun kontekstin, joka sisältää kaikki muuttujat arvoineen, joita voidaan käyttää templaateissa dynaamisten HTML-tiedostojen rakentamisessa. Templaatit voivat myös periä sisältöä toisista HTML-templaateista. Näin käyttäjän ei tarvitse

kirjoittaa HTML-rakennetta suoraan pyynnönkäsittelijään, vaan rakennetta hallitaan omissa HTML-tiedostoissa. (Trudeau 2024.) Kuvassa 27 (n.d.) on esimerkki HTML-templaatista, joka sisältää tiedoston alussa olevan pohjatemplaatin perinnän, kontekstimuuttujien käytön, Jinja2-syntaksista tulevan for-silmukan käytön listarakenteisen kontekstimuuttujan iterointiin ja muuttujien arvojen muuttamiseen tarkoitetut templaattisuodattimet.

KUVA 27. Esimerkki Django-templaatista. Templaatti sisältää osioita, minkä avulla HTML-dokumentti voidaan rakentaa dynaamisesti riippuen siitä, mitä tietoa templaatille annetaan. Kuvan templaatin pohjana käytetään geneeristä pohjatemplaattia, ja siinä on osiot HTML-dokumentin otsikolle ja pääasialliselle sisällölle. (Django n.d.)

Templaatti renderöidään suoraan renderöimällä dynaaminen HTML-templaatti jo etukäteen staattiseksi sivuksi palvelimella käyttäjälle lähettämistä varten (Trudeau 2024). Tästä esimerkkinä kuvan 28 (n.d.) pyynnönkäsittelijä, joka alustaa aluksi kontekstimuuttujan ja renderöi templaatin antamalla sille alustetun kontekstimuuttujan.



KUVA 28. Esimerkki Django-templaatin renderöinnistä index-näkymässä. Näkymäfunktio hakee tietokannasta viisi viimeisintä kysymysobjektia, asettaa ne kontekstiin ja antaa kontekstin templaatille asetettaviksi arvoiksi. (Django n.d.).

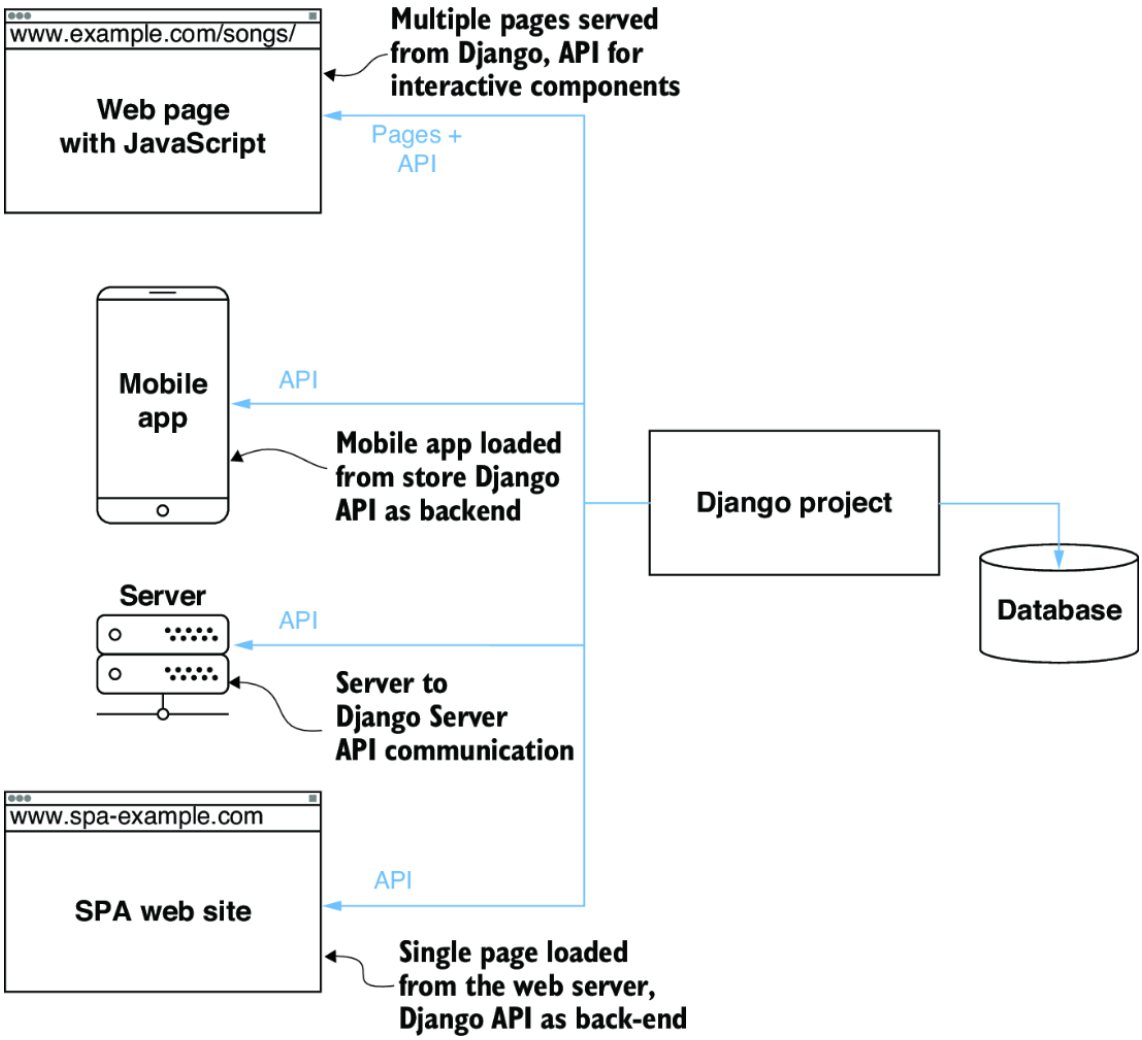
Kuvassa 29 (n.d.) näkyy esimerkkinä kaksi tietomallia. Djangossa voidaan luoda tietomalleja Python-luokkien avulla. Tietomalliluokka edustaa tietokannassa olevaa tietokantataulua ja luokan kentät kuvaavat kyseisen taulun kenttiä (Django, n.d.).

KUVA 29. Esimerkki Djangossa luoduista tietomalleista ja tietomallien kentistä. Kuvan tiedostossa on kaksi tietomallia: Musician ja Album. Tietomalleilta löytyy merkkijono-, päivämäärä- ja numeromuotoisia kenttiä sekä relaatio Album-tietomallista Musician-tietomalliin. (Django n.d.)

Tietokantaan liittyviä muutoksia hallitaan Djangon migraatiotyökalulla. Migraatiotyökalulla luodaan versioituja migraatioita, joilla luodaan, muutetaan ja poistetaan tietokantatauluja. Migraatiot tekevät tietokantamuutoksista loppujen lopuksi SQL-

lausekkeita, joita voidaan ajaa Django-sovelluksen käyttämää tietokannan kanssa. Migraatiotyökalua käytetään sovelluksen ajoajan ulkopuolella, mutta Django tarvitsee tavan keskustella tietokannan kanssa myös ajon aikana. Django käyttää tähän tarkoitukseen objekti-relaatiokartoitusrajapintaa tietokantakutsujen tekemiseen. Tätä rajapintaa kutsutaan yleisesti ORM-rajapinnaksi englanninkielisen vastineen, object relational mapping, mukaan. ORM-rajapinta abstraktoi SQL-kyselyt tietokantaan luokkien ja objektien metodeiksi. Djangon kohdalla ORM-rajapinta luo Python-ohjelmointikielellä QuerySet-objekteja, joiden avulla tietokannan rivejä voidaan hakea ja muokata. (Trudeau 2024.) Kuvassa 28 (n.d.) Question-tietomallin objektit eli tietokantataulun rivit järjestetään laskevaan järjestykseen julkaisupäivän mukaan, ja näistä objekteista otetaan viisi ensimmäistä.

URL-konfiguraatiot, pyynnönkäsittelijät, templaatit, tietomallit ja ORM ovat pohja kaikelle sovelluskehittämiselle Djangossa (Trudeau 2024). Näiden lisäksi Djangoon on sisäänrakennettu laajasti monia ominaisuuksia verkkosovellusten kehittämisen sujuvoittamiseksi, ja tämän takia Djangosta puhutaankin, että siinä tulee mukana kaikki tarvittava.
Django on kuitenkin kehyksenä luonteentaan sellainen, että sen vahvuudet ovat HTML-pohjaisissa verkkosovelluksissa, joissa Django rakentaa HTML-templaattien pohjalta näkymän, jossa käytetään tietokannan tietoja, ja tämä rakennettu HMTL-dokumentti palautetaan takaisin käyttäjälle. Tätä arkkitehtuurimallia, jossa käyttöliittymä rakennetaan kokonaan palvelimella, kutsutaan palvelinpuolen renderöinniksi. Tämän vastakohta on SPA-arkkitehtuurimalli, jossa taustajärjestelmä toimii ensisijaisesti datan välittäjänä käyttäjälle rajapinnan kautta. Data välitetään yleensä SPA-arkkitehtuurimallissa JSON-formaattina selaimelle taustajärjestelmästä. JSON-formaatissa oleva data otetaan vastaan asiakaspuolen sovelluksessa ja käyttöliittymä rakennetaan tämän datan pohjalta JavaScriptillä, jota selain ymmärtää ja ajaa. Logiikka käyttöliittymän rakennuksessa on siis samanlainen kuin Djangossa rakennettu käyttöliittymä HTML-templaattien avulla. Suurin ero on siinä, missä käyttöliittymä rakennetaan. Selaimen lisäksi myös muut alustat voivat hyödyntää tällaista mallia, jossa rajapinta välittää JSON-pohjaista dataa, kuten mobiilisovellukset. (Trudeau 2024.)



KUVA 30. Django-sovelluksen rajapinnan palvelemat sovellukset. Django sovellus tarjoilee useita eri HTML-sivuja, rajapinnan kautta dataa mobiilisovellukselle ja SPA-sovellukselle, ja keskustelee toiselle palvelimelle suoraan. (Trudeau 2024.)

Vaikka Djangon avulla pystyy luomaan sellaisia rajapintoja, joiden avulla pystyy palvelemaan useita alustoja ja palauttamaan JSON-muotoista dataa, joutuu kehittäjä manuaalisti toteuttamaan rajapintoihin esimerkiksi serialisoinnin ja deserialisoinnin, validaatiot, autentikoinnin ja autorisoinnin, reititykset, sivutuksen, versioinnin, virheidenkäsittelyn ja niin edelleen. Tämän takia Django-kehystä käytettäessä onkin mahdollista valita työkalu, joka auttaa juuri tässä asiassa. Hyvin laajasti käytetty työkalu JSON-pohjaisten REST-rajapintoihin luomiseen on Django REST framework -kirjasto, joka tarjoaa etenkin valmiita luokkapohjaisia ratkaisuja ja työkaluja REST-rajapintojen luomiseen. (Trudeau 2024.)

Django REST framework -kehyksen kohdalla Serializer-luokkien tehtävänä on nimensä mukaisesti hoitaa datan serialisointi. Serialisointiluokkien kohdalla serialisoinnille voidaan asettaa myös validointivaatimuksia, joita rajapinta vaatii etenkin vastaantulevalta tiedolta. Etenkin tietomallipohjaiset serialisoijat ovat hyvinkin käteviä työkaluja tietokantaobjektien kanssa työskentelyyn REST-rajapintojen kautta, koska objektien kanssa työskentely, kuten uusien objektien luominen tai olemassaolevien objektien päivittäminen, käy paljon helpommaksi vain asettamalla tarvittavat tietomallikentät serialisoijalle, ja Django REST framework hoitaa tämän pohjalta suurimman osan työstä automaattisesti (Django REST Framework, n.d.)

Serialisointiluokkien päälle rakennetaan ViewSet-luokkia, jotka sisältävät joukon useampia HTTP-metodikohtaisia kutsunkäsittelijöitä. ViewSet-luokkaimplementaatiot sisältävät sovellusten REST-rajapintojen pääasiallisen rajapintatoteutuksien sovelluslogiikan. Rajapinnan kehittäjän täytyy vain tietää, mitkä metodit vastaavat mitäkin HTTP-metodia ja implementoida rajapintalogiikka kyseiseen metodiin. ViewSet-luokkiin voidaan lisätä standardinmukaisten HTTP-metodipohjaisten ViewSet-metodien lisäksi myös ylimääräisiä sovelluskohtaisia metodeja, joihin voidaan toteuttaa yksityiskohtaisempaa logiikkaa. ViewSet-luokat sisältävät myös kenttiä, joilla voi asettaa oikeuksienhallintaa ja autentikaatiotarkistuksia rajapintoihin (Django REST Framework, n.d.)

Routers-luokat sisältävät keskitetysti REST-pohjaisten polkujen reititykset. Reitityksiin määritellään, että millaisia polkuja asiakassovellusten tulee käyttää käyttäessään rajapintaa. Polkuihin asetetaan ViewSet-luokka kutsunkäsittelijäksi. (Django REST Framework, n.a.)

Django REST frameworkin tunnettuudesta huolimatta OpenAPI-ekosysteemin palvelinrunkojen generointityökalut eivät tue Django REST frameworkia eli sitä käyttävät sovelluskehittäjät eivät pysty generoimaan palvelinrunkoja Django REST frameworkin pohjalta rakennetuille sovelluksille ja rajapinnoille (OpenAPI Generator 2024).

Django Ninja on toinen samantyylinen kirjasto kuin Django REST framework, jolla pystyy tekemään REST-pohjaisia rajapintoja. Django Ninja on rakennettu vahvasti FastAPI-nimisen verkkokehyksen vaikutuksesta. FastAPI on myös REST-rajapintoihin tarkoitettu verkkokehys, mutta eroaa Djangosta siten, että FastAPI ei sisällä rajapintaimplementaatioon liittymättömiä osia, kuten autentikaatiota, objekti-relaatiokartoitustekniikkaa tietokantakyselyitä varten tai templaattimoottoria. Tästä huolimatta Django Ninja on rakennettu siten, että se kuitenkin integroituu hyvin osaksi Django-sovellusta, jossa Django Ninja voi toteuttaa JSON-pohjaisen rajapinnan sovellukseen. Django Ninjan sisäinen toteutus on siis muokannut FastAPI-tyylistä rajapintojen toteutusmallia siten, että se menee yhteen Djangon kanssa. On hyvä myös tunnistaa, että Django Ninja ja Django REST framework ovat ohjelmointiparadigmoiltaan erilaisia: Django REST framework on luokkapohjainen, kun taas Django Ninja on hyvin funktiopohjainen FastAPI-kehyjsen mukaisesti (Django Ninja, n.a.)

Django Ninjan hyötynä Django REST frameworkiin verrattuna, että koska Django Ninja on luotu FastAPI-kehyksen pohjalta, voi OpenAPI Generatorilla generoida kehitystä varten FastAPI-pohjaisen pavelinrungon ja vain muokata sitä Django Ninjaan sopivaksi. Django Ninja myös generoi sisäänrakennettuna toimintona OpenAPI-dokumentaation kehitetystä rajapinnasta.

Django Ninjan hyödyistä huolimatta tämä opinnäytetyö keskittyy Django REST framework -kehykseen, koska tarkoituksena on tuottaa arvoa tämän opinnäytetyön toimeksiantajalle eli Haltulle. Opinnäytetyön seuraavassa osassa tuodaan näkyväksi se, miten Django REST framework -pohjaisiin rajapintatoteutuksiin voidaan tuoda OpenAPI-ekosysteemistä työkaluja osaksi kehitystä ja keskittyen erityisesti hyödylliseen ja käytettävään rajapintadokumentaatiogenerointiin.

3. JALKAUTTAMINEN TOIMEKSIANTAJALLE

Luvussa kaksi muodostettiin teoreettinen viitekehys sille rajapintoihin sekä niiden kuvaamiseen ja dokumentointiin liittyviin aihealueisiin ja ongelmakenttiin. Tätä tietoa lähdettiin hyödyntämään konkreettisesti teknisenä toteutuksena osana Haltun kehitysprosesseja. Tämä luku sisältää taustoituksen, mistä toteutusta lähdettiin luomaan, sekä teknisen kuvauksen itse toteutuksesta.

3.1. Opinnäytetyön ja toteutuksen taustoitus

OpenAPI-dokumentaatio nousi nopeasti ehdotuksena Haltulta opinnäytetyön aiheeksi, kun keskustelu konkreettista arvoa tuottavasta opinnäytetyöstä nostettiin puheenaiheeksi. Haltun kehittäjillä oli kokemusta historiasta sovelluksien kanssa työskentelystä, joiden rajapintadokumentaatio ei ole ollut hyödyllistä kehityksen tueksi. Yhdeksi esimerkiksi huonosta dokumentaatiosta nousi yksityiskohta yhdestä kolmannen osapuolen sovelluksesta, jonka rajapinta kuvasi päätepisteen vastaanottavaa parametria nimellä *data*, mutta dokumentaatio ei kuvannut mitenkään tarkemmin, mitä kyseinen parameteriarvo oikeastaan tarkoittaa. Rajapinnan kehittäjät olivat siis luoneet OpenAPI-spesifikaation pohjalta dokumentaatiota omasta rajapinnastaan, mutta sen hyöty jäi vähäiseksi vajavaisen dokumentaatiokuvauksen takia. Pelkkä tekniikka automaattisesta rajapintakuvauksesta ei siis riitä vaan dokumentaatiolle täytyy antaa ajatusta, jotta kehittäjät voivat käyttää sitä oikeasti hyödykseen. Samaista haastetta on myös ollut Haltun itse kehittämien julkisten rajapintojen kanssa, joita voidaan käyttää ulkoisten tahojen toimesta. Tarvetta etenkin hyödyllisen rajapintadokumentaation luomiseen löytyi. Tähän tarpeeseen lähdettiin etsimään ratkaisua Haltulle, jotta samoja turhauttavia ja kehitystä vaikeuttavia tilanteita historiasta voitaisiin välttää tulevaisuudessa.

3.2. Mallipohjien hyödyntäminen toteutuksessa

Haltulla on jo pitkään ollut käytössä projektien rakentamiseen työkaluna lähdekoodirepositorioon mallipohjia, joita Haltulla kehittäjät voivat käyttää eri tarkoituksia varten. Mallipohjat ennen kaikkea dokumentoivat, miten Haltulla toteutetaan projekteja standardoidusti, joten kehittäjät voivat tarpeen tullen tarkistaa mallipohjista, miten sovelluksen pohjatoteutuksessa eri osioita on rakennettu. Mallipohjat myös nopeuttavat radikaalisti uusien projektien aloittamista, koska kehittäjien ei tarvitse uusien projektien alkaessa aloittaa projektin konfiguraatioiden, kehitysympäristön ja dokumentaation kirjoittamista aina alusta. Kehittäjät voivat tällöin nopeasti siirtyä projektin alkaessa siihen, mikä tuottaa arvoa projektille.

Tällä hetkellä projektin lähdekoodin mallipohjat eivät sisällä DRF-pohjaisen toteutuksen automaattisen rajapintadokumentaation implementaatiota. Myöskään OpenAPI-dokumenttia ei löydy ennestään mallipohjasta. Tämä tarkoittaa sitä, että sovelluskehittäjä Haltulla joutuu aina uuden projektin alkaessa toteuttamaan alusta asti sovelluksen rajapinnalle automaattisen dokumentaation toteutuksen. Opinnäytetyössä lähdettiin täyttämään tätä puutetta mallipohjasta. Tämän opinnäytetyön kohdalla ei tulla tekemään seurantaa kehitetyn mallipohjan käytöstä, mutta koska mallipohja on aktiivisesti käytössä Haltulla, hypoteesi on, että Haltulla tulee olemaan suhteellisen kivutonta, koska uudet projektit, jotka käyttävät mallipohjaa, tulevat jatkossa automaattisesti sisältämään rajapintakuvauksen ja -dokumentoinnin hyödyt. Olemassa olevien projektien on myös helppo toteuttaa automaattisen rajapintadokumentaation, sillä mallipohja toimii jatkossa dokumentoituna referenssinä.

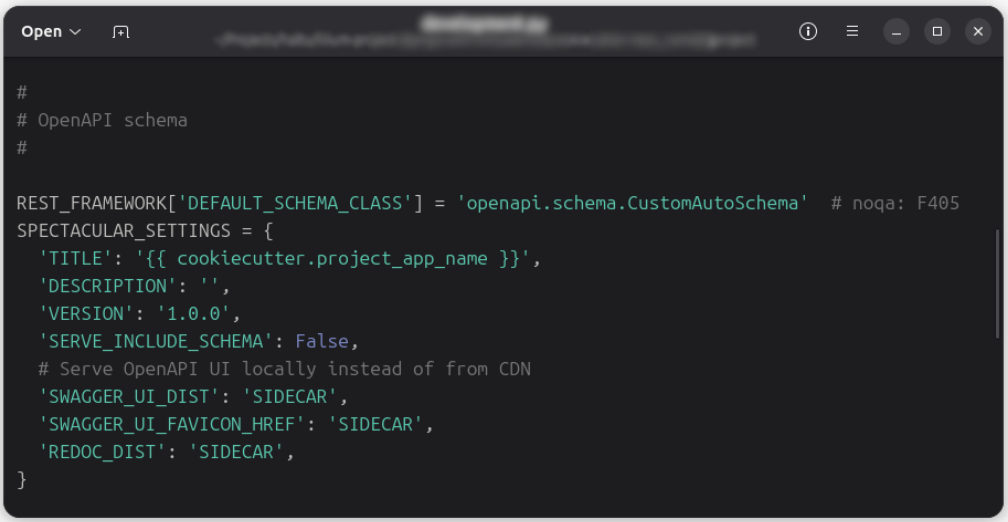
3.3. Toteutuksen tekninen kuvaus

Mallipohjaan lisättiin kaksi kokonaisuutta: Django REST framework -kehys rajapinnan esimerkkitoteutusta varten, josta rajapintadokumentaatio luodaan, ja rajapinnan automaattisen dokumentaatiosivun luominen drf-spectacular-kirjaston avulla käyttäen Swagger UI - ja Redoc-dokumentointityökaluja. Pienin mahdollinen toteutus rajapinnan automaattista dokumentointisivua luontia varten olisi ollut pelkän drf-spectacular-konfiguraation lisääminen mallipohjaan, mutta myös Django REST frameworkin avulla tehtävä rajapintatoteutus koettiin tarpeelliseksi,

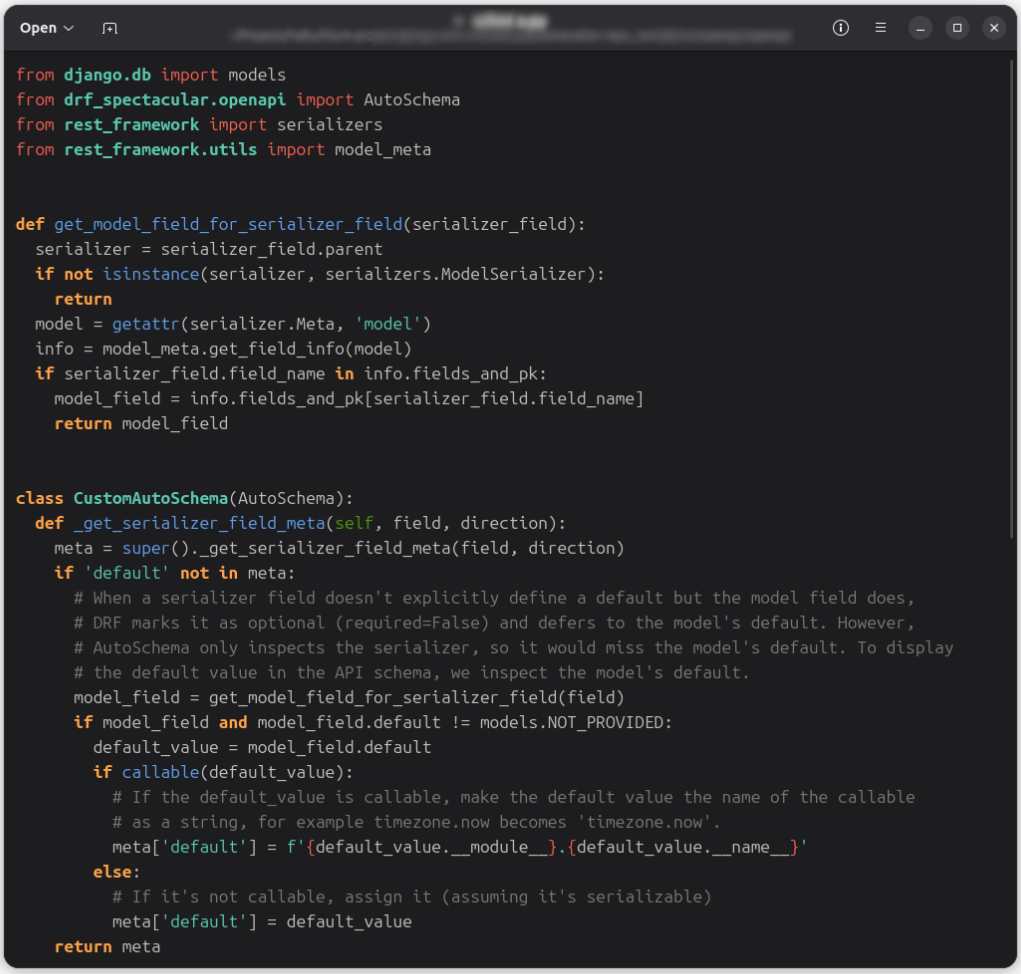
jotta mallipohja sisältää myös esimerkkejä siitä, miten OpenAPI-skeemaa voidaan laajentaa paremman dokumentaatiokokemuksen varmistamiseksi. Automaattisen OpenAPI-skeeman ja dokumentaatiosivun toteuttava drf-spectacular-kirjasto on Django REST framework -kehyksen (n.d.) suosittelema valinta rajapinnan dokumentointia varten, joten kirjaston valinta mallipohjaan oli luonnollinen valinta.

Nimensä mukaiset drf-spectacular hyödyntää Django REST frameworkin kehyksellä luotuja, APIView-luokan periviä luokkapohjaisia kutsunkäsittelijöitä. Django REST frameworkilta löytyy AutoSchema-luokka, joka käy läpi kaikki Django-sovelluksen kutsunkäsittelijät ja tarkastelee, että onko käsittelijä luokka ja APIView-pohjainen. AutoSchema-luokka voidaan myös ylikirjoittaa, jos sovelluksella on tarvetta mukauttaa AutoSchema-luokan oletustoimintapaa. Tämä tapahtuu automaattisesti taustalla Django REST frameworkin tarjoaman DEFAULT_SCHEMA_CLASS-konfiguraatiomuuttujan kautta. Oletuksena muuttujan arvo on rest_framework.schemas.openapi.AutoSchema eli rajapinnasta luodaan OpenAPI-skeema, joka on tällä hetkellä alan käytetyin rajapinnankuvausskeema, mutta myös muitakin skeemoja on mahdollista luoda. Muita skeematyyppejä ei kuitenkaan käydä läpi tässä opinnäytetyössä.

Kuvassa 31 on mallipohjaan lisätyt konfiguraatiot OpenAPI-skeemaa ja rajapintadokumentaatio varten. Mallipohjaan on luotu oma openapi-niminen Django-sovellus, joka rajaa toiminnallisuutensa OpenAPI-skeeman ja dokumentaatiosivujen automaattiseen luomiseen. Kyseisestä sovelluksesta löytyy mukautettu AutoSchema-luokka, jonka tehtävänä on tarjota ratkaisu ongelmaan drf-spectacular-luokan kanssa. Kuvassa 31 on kuvattu kyseinen ratkaisu: vaikka Django REST frameworkin serialisoijaluokat osaavatkin hyödyntää tietomallien oletusarvoja oikein, vaikka tietomallin kentän oletusarvoa ei erikseen lisätä serialisoijaluokalle, tätä oletusarvoa ei lisätä automaattisesti luotuun OpenAPI-skeemaan. Mukautetussa luokassa on toiminto, joka hakee oletusarvon skeemalle suoraan tietomallilta näissä tapauksissa.



KUVA 31. Automaattisen OpenAPI-skeeman ja dokumentaatiosivun luomisen konfiguraatio. Konfiguraatio on OpenAPI-skeema otsikon alla. Konfiguraatio sisältää DEFAULT_SCHEMA_CLASS-muuttujan arvon sekä SPECTACULAR_SETTINGS-konfiguraatiot drf-spectacular-kirjastolle.



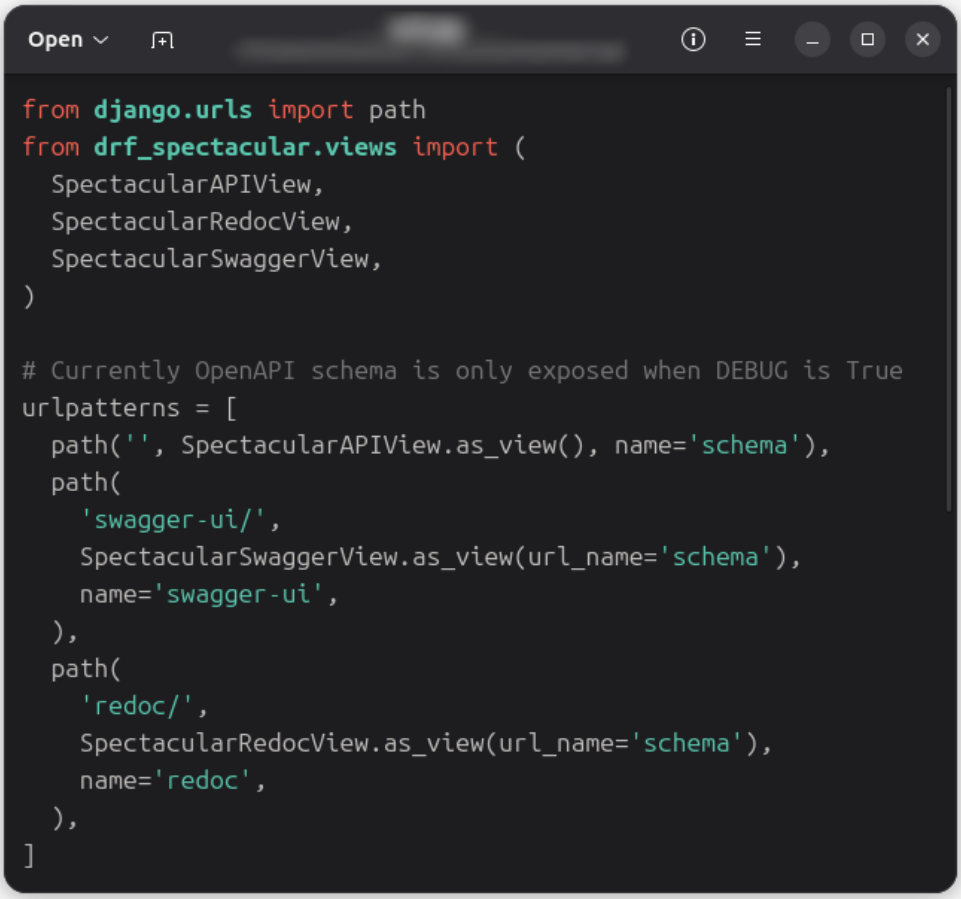
```python
from django.db import models
from drf_spectacular.openapi import AutoSchema
from rest_framework import serializers
from rest_framework.utils import model_meta


def get_model_field_for_serializer_field(serializer_field):
  serializer = serializer_field.parent
  if not isinstance(serializer, serializers.ModelSerializer):
    return
  model = getattr(serializer.Meta, 'model')
  info = model_meta.get_field_info(model)
  if serializer_field.field_name in info.fields_and_pk:
    model_field = info.fields_and_pk[serializer_field.field_name]
    return model_field


class CustomAutoSchema(AutoSchema):
  def _get_serializer_field_meta(self, field, direction):
    meta = super()._get_serializer_field_meta(field, direction)
    if 'default' not in meta:
      # When a serializer field doesn't explicitly define a default but the model field does,
      # DRF marks it as optional (required=False) and defers to the model's default. However,
      # AutoSchema only inspects the serializer, so it would miss the model's default. To display
      # the default value in the API schema, we inspect the model's default.
      model_field = get_model_field_for_serializer_field(field)
      if model_field and model_field.default != models.NOT_PROVIDED:
        default_value = model_field.default
        if callable(default_value):
          # If the default_value is callable, make the default value the name of the callable
          # as a string, for example timezone.now becomes 'timezone.now'.
          meta['default'] = f'{default_value.__module__}.{default_value.__name__}'
        else:
          # If it's not callable, assign it (assuming it's serializable)
          meta['default'] = default_value
    return meta
```

KUVA 32. Mukautettu AutoSchema-luokka. Kuvassa on Python-moduuli, joka sisältää CustomAutoSchema-nimisen luokan, joka perii AutoSchema-luokasta. CustomAutoSchema yliajaa _get_serializer_field_meta-metodin ratkaistakseen tietomallin kenttien oletusarvoihin liittyvän ongelman AutoScheman kanssa.

Yleisten skeema-asetusten lisäksi konfiguraatioissa on kaksi muuta muutosta. Skeeman omaa rajapintapolkua ei lisätä mukaan skeemaan sekä tämän myötä ei myöskään rajapintadokumentaatioon. Rajapintadokumentaatiosivut toteuttavat staattiset tiedostot jaellaan sisällönjakeluverkoston sijaan lokaalisti samasta ympäristöstä kuin sovelluksen muut tiedostot, jotta dokumentaatiot ovat aina saatavilla riippumatta siitä, onko verkkoyhteyttä saatavilla.

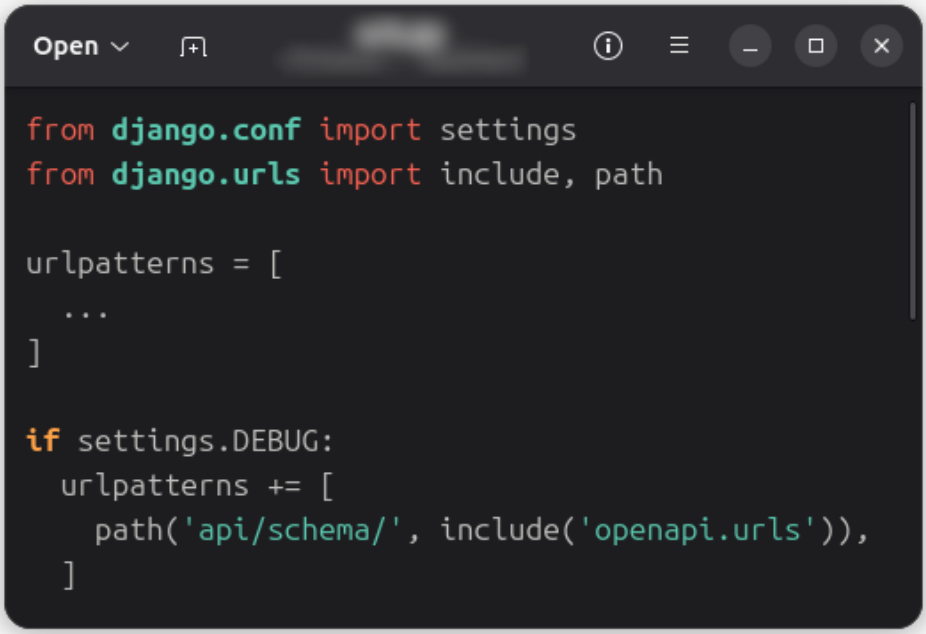
Staattisien tiedostojen asetuksista voi nähdä, että mallipohja tukee jatkossa sekä Swagger UI -työkalun että vaihtoehtoisen ReDoc-työkalun dokumentaatiosivuja. Kehittäjä voi itse valita kumpi dokumentaatiotyyleistä on mieluisampi.

Työkalujen luomalla rajapinnan dokumentaatiosivulla voi käydä niille määriteltyjen polkujen kautta, kuten kuvassa 33 on määritelty.



KUVA 33. Skeeman ja rajapintadokumentaatiosivujen polkumäärittelyt. Skeema löytyy openapi-sovellukselle määritellyn polun juuripolusta. Swagger UI -sivun polku on swagger-ui/ ja ReDoc-sivun polku on redoc/.

Skeema- ja dokumentaatiopolut, jotka määritellään openapi-sovelluksessa, yhdistetään lopuksi koko projektin tasolle, jotta polut ovat löydettävissä sovelluksen osoitteen kautta. Tätä yhdistämistä ei tehdä tosin tuotantoympäristössä. Tämä sen takia, että dokumentaatiosivuille on ollut tällä hetkellä tarvetta Haltulla erityisesti kehityksen tueksi ja koska rajapinnat ovat sovelluksissa usein sisäisiä ja käyttöliittymää varten tehtyjä. Jos rajapinnan tulee olla julkinen, projektin kannattaa tällöin poistaa rajoite kehitysympäristökäytöstä.

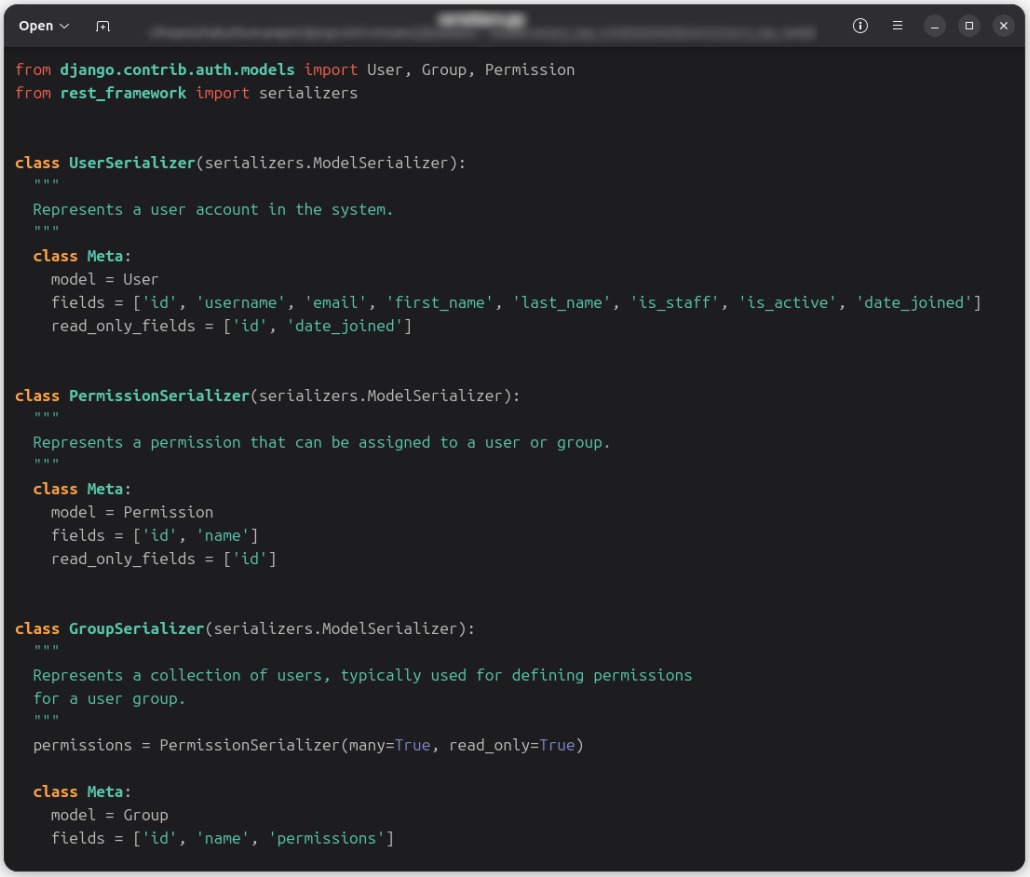


KUVA 34. Openapi-sovelluksen polkumäärittelyjen yhdistäminen projektitason muihin polkuihin. Yhdistäminen tehdään muihin polkuihin vain, jos projektin asetuksiin määritelty DEBUG-muuttuja on arvoltaan True. Openapi-sovelluksen pohjapoluksi on määritelty api/schema/.

Näiden määrittelyjen jälkeen openapi-sovellusta koskevien määrittelyjen jälkeen mallipohjan on jo mahdollista luoda rajapintadokumentaatiosivu, mutta sivulla ei vain ole vielä tässä näy vielä rajapinnasta mitään tietoja, koska rajapintaa ei ole vielä määritelty. Tämän takia mallipohjaan määriteltiin esimerkkirajapinta valmiiksi, jotta dokumentaatiosivulla näkyy rajapintadokumentaatiota ja kehittäjä näkee mallipohjasta suoraan, miten drf-spectacular-kirjastoa tulisi käyttää rajapintojen hyödylliseen dokumentointiin.

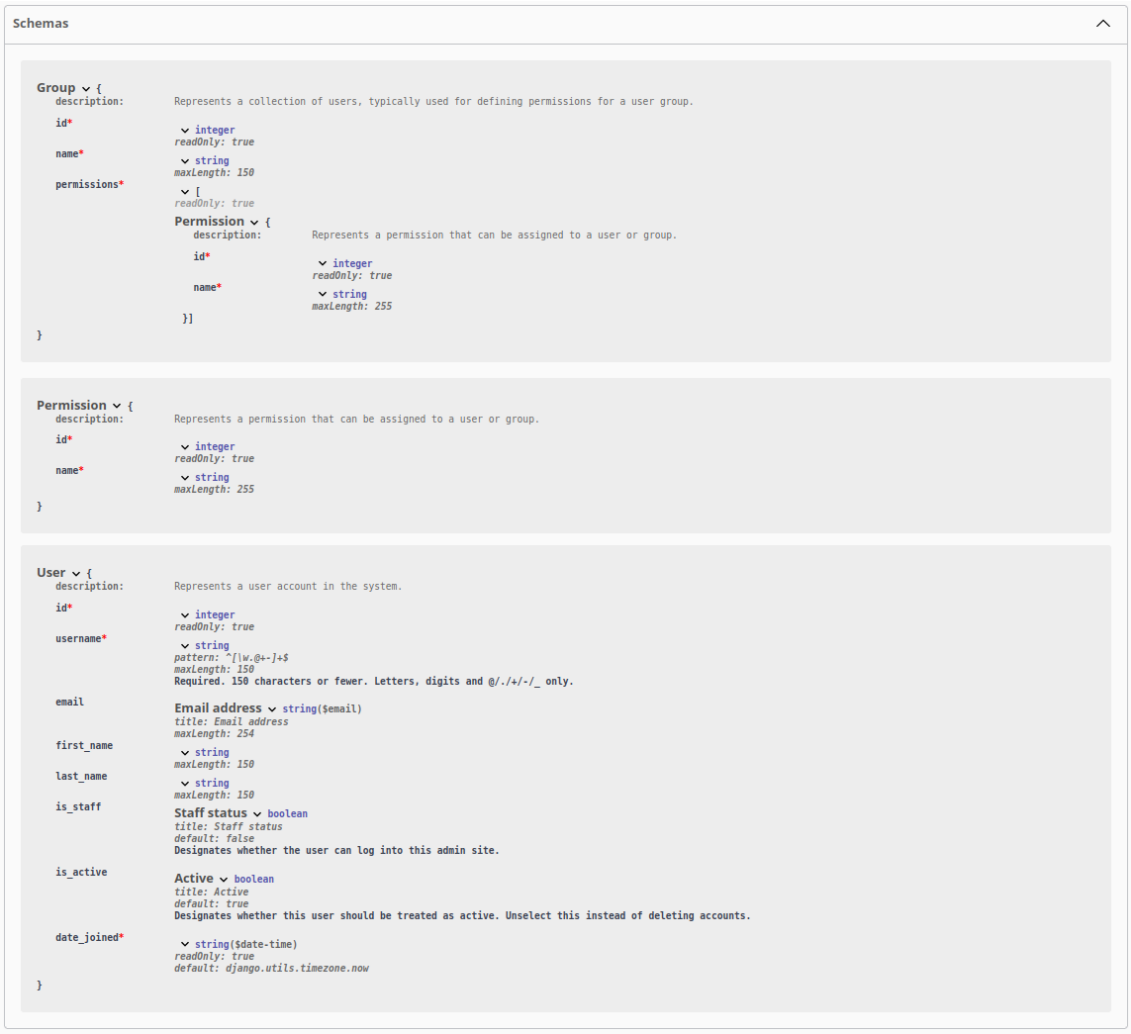
Rajapintadokumentaatiosivuilla näkyy tietoa sekä rajapinnan päätepisteistä että rajapinnan käsittelemistä skeemoista, joita sovelluksen rajapinta julkaisee rajapinnan kautta jaettavaksi asiakkaille. Swagger UI ja ReDoc esittävät näitä tietoja hieman eri tavalla, mutta molemmat tiedot ovat saatavilla ja ne koetaan myös Haltun työntekijöiden näkökulmasta tärkeäksi.

Skeemojen tiedot tulevat dokumentaatioon Django REST frameworkin serialisoijaluokkien kautta. Sovellus voi käyttää erityyppisiä Serializer-luokkia. Usein käytössä on ModelSerializer-luokka, joka julkaisee Djangon avulla luodun tietomallin, joka kuvastaa tietokannan tauluja Python-luokkina, rajapinnan kautta, mutta kehittäjä voi julkaista rajapinnan päätepisteen, jonka toiminta ei perustu tietokantatauluun. Mallipohjan kohdalla käytämme ModelSerializer-luokkia. Koska kyseessä on mallipohja, emme etukäteen tiedä, minkä tyyppisiä tietomalleja projekti tulee tarvitsemaan, joten halusimme käyttää ainoastaan sellaisia tietomalleja, joita mallipohja sisälsi etukäteen, jotta emme turhaan tuo kehittäjien vaivaksi uusia tietomalleja, joita jouduttaisiin aina poistamaan. Tämän myötä päädyimme käyttämään Djangon sisäisiä tietomalleja django.contrib.auth-paketista, joita ovat Group-, Permission- ja User-tietomallit. Group on käyttäjien ryhmittelyyn tarkoitettu tietomalli, joka provisioi kaikille ryhmän käyttäjille ennaltamääriteltyjä käyttöoikeuksia. Permission-tietomalli edustaa näitä käyttöoikeuksia. User-tietomalli edustaa palvelun käyttäjiä. Tietomalleihin sidotut serialisoijat on määritelty kuvan 35 mukaisesti. PermissionSerializerin tehtävä ei ole määritellä serialisoitavia kenttiä Permission-tietomallin omaa päätepistettä varten. Group-tietomallilla on monen suhde moneen -relaatio Permission-tietomalliin, joten PermissionSerializerin tehtävä on määritellä yksittäisen Permission-objektin muoto, jota esitetään ryhmien päätepisteen kautta.



KUVA 35. Tietomallien serialisoijaluokat. Python-moduulissa on kolme serialisoijaa: UserSerializer, PermissionSerializer ja GroupSerializer. GroupSerializer käyttää PermissionSerializeria permissions-kentän serialisoijana.

Serialisoijaluokkien myötä rajapintadokumentaation skeemojen osuus näyttää Swagger UI:n käyttöliittymässä kattavalta ja siinä näkyy myös serialisoijaluokkien koodimäärittelyjen lisäksi tärkeitä validaatio- ja vaatimustietoja suoraan tietomallien kenttämäärittelyistä. Serialisoijaluokkien pydoc-merkkijonot dokumentoivat skeemoja entisestään erillisillä kuvausteksteillä. (kuva 36.)



KUVA 36. Swagger UI:n skeemojen osio. Skeemoissa näkyy Group-, Permission- ja User-skeemat. Jokaisen skeeman kenttämäärittelyt on avattu, joista näkyy kenttien tarkat tiedot sekä skeeman kuvausteksti.

Serialisointiluokkien ja skeemojen dokumentoinnin jälkeen jäljelle jäi enää rajapinnan päätepisteiden määrittelyt. User-tietomalliin liittyvä päätepiste määriteltiin UserViewSetiksi ja Group-tietomalliin liittyvä päätepiste GroupViewSetiksi. GroupViewSet on lisätty esimerkiksi kuvaan 37.

KUVA 37. Group-tietomallin päätepistemäärittelyt GroupViewSetillä. GroupViewSetiin on määritelty list- ja retrieve-metodit kaikkien sekä yksittäisten Group-objektien hakemiseksi GET-metodilla. Metodeille on lisätty ylimääräisiä skeemamäärittelyitä @extend_schema-dekoraattorilla.

Koska list- ja retrieve-metodit itsessään ei sisällä yhtään mukautettua logiikkaa suhteessa perityn Django REST frameworkin viewsets.ReadOnlyModelViewSet-luokkaan, oleelliseksi osaksi päätepisteiden määrittelyä jää @extend_schema-dekoraattorin sisältö. Drf-spectacular-kirjasto tarjoaa paljon ennaltamääriteltyjä arvoja automaattisesti luotuun skeemaan, mutta drf-spectacularin ennaltamäärätyt arvot eivät aina tuota haluttua dokumentointia. Summary-parametri määrittää päätepisteen lyhyen kuvauksen, mikä näkyy Swagger UI:ssa päätepistetietojen ollessa kutistettuna. Oletuksena tämä kuvaus on tyhjä, mutta mallipohjassa näille

halutaan antaa arvot. Description-parametrin avulla määritellään päätepisteen pitempi kuvausteksti. Oletuksena kuvausteksti on tyhjä tai sitten kuvausteksti otetaan luokan pydoc-merkkijonosta. Tämä on parempi kuin kokonaan tyhjä kuvausteksti, mutta jos kyseessä on useamman päätepisteen luokka, tulee tällöin sama kuvausteksti jokaiselle päätepisteelle. Etenkin niissä tapauksissa, missä saman kutsunkäsittelijöiden luokan metodien välillä on erityisiä eroja, on nämä tärkeä dokumentoida päätepisteen kuvaukseen. Kolmas parametri koskee rajapintadokumentaation esimerkkikutsuja ja -kuvauksia. Esimerkit dokumentaatiossa ovat tärkeitä, jotta kehittäjät saavat nopeasti kuvan, millä tavalla rajapinnan päätepisteiden kanssa. Rajapintadokumentaatiot sisältävät esimerkkejä oletuksena, mutta esimerkkiobjektien kentillä ei ole kunnon esimerkkiarvoja, vaan kenttien kohdalla lukee niiden tyypit. Oikeilla esimerkkiarvoilla kehittäjä saa paremman kuvan, minkälaista dataa rajapinta tarjoaa, ei vain datan tyyppejä.

Nämä päätepisteet lisätään lopuksi osaksi muun Django-sovelluksen polkujen määrittelyä, jotta päätepisteisiin voidaan tehdä kutsuja. Django REST framework hoitaa tämän Router-luokkien kautta, joiden avulla päätepisteille annetaan aloituspolku, minkä pohjalta lopulliset polut rakennetaan REST-arkkitehtuurityylin mukaisesti. Mallipohjaan nämä reitit tehtiin kuvan 38 mukaisesti.

KUVA 38. UserViewSet- ja GroupViewSet-päätepisteluokkien reitit. Moduulissa alustetaan DefaultRouter-luokka, johon reitit tehdään. Reitit aloitetaan api/-polusta.

OpenAPI-skeeman ja rajapintadokumentaation luomisen kokonaisuus saatiin näillä muutoksilla sellaiseen vaiheeseen, josta on hyvä aloittaa OpenAPI-skeeman ja rajapintadokumentaation integroimisen osaksi sovellusten kehittämistä. Tästä toteutuksesta seuraa Haltulla sisäistä seurantaa toteutuksen arvontuotosta sekä sen merkityksestä ja kehittämiskohteista.

4. POHDINTA

4.1. Yhteenveto

Opinnäytetyön keskeisenä tavoitteena oli vastata Haltu Oy:n tunnistamaan haasteeseen liittyen rajapintadokumentaation olemassaoloon sekä laatuun. Kokemukset huonosti dokumentoiduista rajapinnoista, niin kolmansien osapuolten kuin Haltun omienkin kehittämien rajapintojen osalta, korostivat tarvetta selkeämmälle ja hyödyllisemmälle dokumentaatiolle.

Ratkaisuna tähän esiteltiin OpenAPI-spesifikaatioon pohjautuvan rajapintadokumentaation toteutus osana Haltun Django-projektimallipohjia. Toteutustapa perustui Django REST framework -kehykseen ja drf-spectacular-kirjastoon, sillä Django REST framework on Django-ekosysteemin standardiratkaisu REST-rajapintojen tekemiseen. Näiden työkalujen ympärille oli mahdollista automaattisesti generoida OpenAPI-skeema ja dokumentaatiosivut suoraan sovelluksen Django REST frameworkin avulla luoduista rajapinnoista. Tämä toteutus tarjosi selkeää parannusta aiempaan tilanteeseen, jossa mallipohjasta puuttui rajapintadokumentaation toteutus kokonaan.

Opinnäytetyössä luotu ratkaisu ei rajoittunut pelkkään tekniseen toteutukseen, vaan pyrki varmistamaan dokumentaation hyödyllisyyden. Esimerkkirajapinnan ja mukautetun AutoSchema-luokan avulla varmistettiin, että dokumentaatioon sisältyy kattavasti metadataa, kuvauksia ja realistisia esimerkkejä, jotka ovat tärkeitä rajapintojen kuluttajille. Opinnäytetyöprosessin alussa jo tunnistettiin Haltun kanssa, että automaattinen generointi on vasta ensimmäinen askel. Varsinainen arvo syntyy dokumentaation sisällön laadusta ja sen soveltuvuudesta kehittäjien tarpeisiin, ja siinä opinnäytetyö onnistui.

Teknisesti toteutetun mallipohjaratkaisun lisäksi opinnäytetyö kokosi rajapintoihin ja OpenAPI-ekosysteemiin liittyvää teoreettista kokonaisuutta, jota voidaan hyödyntää Haltulla rajapintakehitykseen liittyvän osaamisen kasvattamiseen.

4.2. Analyyttinen arviointi

Vaikka varsinaista seurantatutkimusta kehitetyn mallipohjan käyttöönotosta ja sen tuottamasta arvosta ei toteutettu opinnäytetyön puitteissa, odotetaan sen tuovan kehitystyöhön niitä hyötyjä, mitä siltä odotettiin, kun työ tilattiin.

Työssä tunnistettiin dokumentaation laadun tärkeys osana toteutusta, jolloin toteutus ei jäänyt puutteelliseksi, jossa vain automaattinen dokumentointi mahdollistettiin. Laadun ylläpitäminen ja tiedon rikastaminen haluttiin sisällyttää osaksi toteutusta. Työssä keskityttiin tekniseen ja sisällölliseen mahdollistamiseen, mutta prosessit jäivät huomioimatta. Haltulla esimerkiksi käytetään käytetään usein tarkistuslistoja osana laadunvarmistuksen prosessia laadun kasvattamiseksi. Tarkistuslistoihin olisi voinut lisätä uuden tarkistuskohdan, joka varmistaa sitä, että rajapintadokumentaation laatua ylläpidetään kehityksen aikana jatkuvasti. Tarkistuslistat tällöin myös muistuttaisivat työntekijöitä dokumentaation tärkeydestä. Kyseiset tarkistuslistat ovat versiohallittuna osana mallipohjarepositoriota, joten tämä olisi ollut helppo lisätä osaksi muutoksia.

Hyötyjen arvioimiseksi Haltun olisi hyödyllistä selvittää sekä ylläpitää kehitetyn mallipohjan tuomaa hyötyä. Rajapintadokumentaation tuomia hyötyjä voidaan selvittää laadullisilla tutkimuksilla työntekijöitä haastatellen. Käyttöönoton varmistamiseksi koulutusten järjestäminen opinnäytetyön tiedollisen kokonaisuuden ja teknisen puolen osalta olisi myös kannatettavaa.

4.3. Jatkotutkimus- ja kehitysideat

Tämän opinnäytetyön aiheille löytyy useita eri jatkokehitys- ja jatkotutkimusaiheita. Ennen opinnäytetyön aloittamista OpenAPI-rajapintakuvausten hyödyntäminen kohdistui lähes pelkästään rajapintadokumentaatioon ja sen luomisen automatisointiin. Opinnäytetyön myötä ymmärrys OpenAPI-kuvasten ympärille luodusta ekosysteemistä kasvoi huomattavasti. Tämä ymmärrys mahdollistaa merkittäviä mahdollisuuksia opinnäytetyön myötä mallipohjan kehittämisen jatkolle. Esimerkiksi SDK-kirjastojen ja palvelinrunkojen automaattisen luonnin integrointi osaksi Haltun kehitysprojektissa voisi olla luonnollinen seuraava kehityskohde

kasvattamaan kehitystyön nopeutta ja laatua. OpenAPI-ekosysteemin nykyisissä työkaluissa on vielä se haaste, että Django- ja Django REST framework -kehyksille ei ole palvelinrunkotyökaluja. FastAPI- ja Flask-kehyksille, jotka ovat kevyempiä ja pienempiä kehyksiä verkkosovellusten ja etenkin rajapintojen tekemiseen Djangoon verrattuna, löytyy palvelinrunkotuki. Arvio palvelinrunkojen sekä muiden työkalujen tuomista hyödyistä voi täten ohjata sitä valintaa, mitä teknologioita käytetään kehityksessä. SDK-kirjastojen tuen pilotointi sekä palvelinrunkojen hyötyjen ja tuettujen teknologioiden arviointi olisivat konkreettisia toimia, mitä Haltu voisi tehdä.

Julkisten rajapintojen ympärille luodaan nykyään hyvin laajasti liiketoimintaa. Tämä tarkoittaa sitä, että jos rajapinta ja sen tarjoamat palvelut ja resurssit ovat itsessään tuote, mitä myydään, tulee rajapintatuotteelle erityisiä vaatimuksia sen julkisen luonteen vuoksi. Sen takia sekä OpenAPI-ekosysteemin työkalut että yleisesti rajapintojen monitorointi- ja hallintatyökalut ovat tällaisessa liiketoiminnassa tärkeitä, koska ne varmistavat kyseisen liiketoiminnan jatkuvuutta ja asiakastyytyväisyyttä. Tämä pitää sisällään myös Haltulla jo aloitetun rajapintadokumentaation kehittämisen. Mitä pidemmälle Haltulla siis viedään tukea rajapintojen kehittämiseen ja hallintaan, voi Haltu luoda uutta sen osaamisen ympärille uutta liiketoimintaa.

Opinnäytetyön teon yhteydessä Haltulla keskusteltiin myös skeema ensin -lähestymistavasta ja sen tutkimisesta osana sovelluskehitystä. Tällaisessa lähestymistavassa järjestelmän osien, kuten rajapintojen ja tietokantojen, skeemat eli rakenteet suunnitellaan ensin ja vasta sen jälkeen kehitetään. Tällä pyritään kasvattamaan yhteistä ymmärrystä ja sitoutumista yhteisesti sovittuun sopimukseen. Haltun kehittäjät ovat kuitenkin huomanneet, että usein kehitystyö kuitenkin ohjautuu enemmän sen mukaan, että mitä tarpeita kehityksen yhteydessä tulee eteen. Erityisesti rajapintaan tulee tarpeita käyttöliittymän puolelta, eikä niin päin, että rajapinnan kautta määritellään se, mitä taustajärjestelmä tarjoaa ja käyttöliittymä mukautuu siihen. Tämän taustan pohjalta olisi hyvä tutkia skeema ensin -lähestymistavan hyötyjä kehitystyössä. Joissakin kehityskielissä skeema ensin -lähestymistavan mukaiset käytännöt ovat paljon pidemmällä kuin toisissa. Java on tästä hyvä esimerkki. Näiden kehityskielien tarjoamia työkaluja ja tapoja skeema ensin

-lähestymistavan toteuttamiseen olisi myös tarpeellista tutkia tällaista aihetta käsitellessä. Haaste skeema ensin -lähestymistavassa on se, että se vaatii kehittäjiltä erillistä osaamista rajapintojen suunnitteluun ja sen kautta yleensä myös tietokannan suunnitteluun, jos rajapinta julkaisee päätepisteitä tietomalleista. Tämä opinnäytetyö ei käsitellyt ollenkaan tätä puolta, että miten rajapintoja tulisi suunnitella. Opinnäytetyön teoreettinen viitekehys tarjosi yleisen katsauksen rajapintoihin sekä HTTP-protokollaan ja REST-arkkitehtuurityyliin, jotka toimivat rajapintatoteutuksien mahdollistajina, mutta se prosessi, miten rajapinta syntyy, ei oteta tässä huomioon. OpenAPI-ekoysteemin työkalut toki kuuluvat tähän samaiseen prosessiin. Tätä prosessia voitaisiin avata rajapinnan koko elinkaarenosalta suunnittelusta alkaen.

LÄHTEET

Tietotekniikan termitalkoot. 2014. Hakemistot. Verkkosivu. Viitattu 5.9.2023. https://sanastokeskus.fi/tsk/fi/termitalkoot/haku-266.html
f
Yhdysvaltain korkein oikeus. 2023. Google LLC vastaan Oracle America, Inc. Valitus Yhdysvaltojen muutoksenhakutuomioistuimelle. Pdf-dokumentti. Viitattu 5.9.2023. https://www.supremecourt.gov/opinions/20pdf/18-956_d18f.pdf

Yhdysvaltain piirituomioistuin Columbiassa. 2000. Lopullinen tuomio. Verkkosivu. Viitattu 5.9.2023. https://www.justice.gov/atr/final-judgment-us-v-microsoft-corporation-state-new-york-et-al-v-microsoft-corporation

Amazon Web Services. n.d. What Is An API (Application Programming Interface)? Verkkosivu. Viitattu 5.9.2023. https://aws.amazon.com/what-is/api/

Jin B., Sahni S. & Shevat A. 2018. Designing Web APIs. O'Reilly Media, Inc.

IBM n.d. What is an API? Verkkosivu. Viitattu 10.9.2023. https://www.ibm.com/topics/api

Gough J., Bryant D. & Auburn M. 2022. Mastering API Architecture. O'Reilly Media, Inc.

Ponelat J. & Rosenstock L. 2022. Designing APIs with Swagger and OpenAPI. Manning Publications.

Pollard, B. 2019. HTTP/2 in Action. Manning Publications.

MDN. 2023. What is a web server? Verkkosivu. Viitattu 18.9.2023 https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/What_is_a_web_server

Waterloon yliopisto. n.d. Client-server Architecture. Pdf-dokumentti. Viitattu 18.9.2023. https://cs.uwaterloo.ca/~m2nagapp/courses/CS446/1195/Arch_Design_Activity/ClientServer.pdf

Netcraft. 2023. August 2023 Web Server Survey. Verkkosivu. Viitattu 18.9.2023. https://www.netcraft.com/blog/august-2023-web-server-survey/

Gourley D., Totty B., Sayer M., Aggarwal A. & Reddy S. 2002. HTTP: The Definitive Guide. O'Reilly Media, Inc.

Grigorik I. 2017. HTTP protocols. O'Reilly Media, Inc.

W3Techs. 2023. Usage statistics of site elements for websites. Verkkosivu. Viitattu 19.9.2023. https://w3techs.com/technologies/overview/site_element

Nginx. n.d. Allowed HTTP Methods. Verkkosivu. Viitattu 24.9.2023. https://docs.nginx.com/nginx-management-suite/acm/how-to/policies/allowed-http-methods/

Bishop, M. 2022. RFC 9114. HTTP/3. Verkkosivu. Viitattu 19.9.2023. https://datatracker.ietf.org/doc/html/rfc9114

Fielding R., Nottingham M. & Reschke J. 2022. RFC 9112. HTTP/1.1. Verkkosivu. Viitattu 23.9.2023. https://datatracker.ietf.org/doc/html/rfc9112/

Fielding R., Nottingham M. & Reschke J. 2022. RFC 9110. HTTP Semantics. Verkkosivu. Viitattu 23.9.2023. https://datatracker.ietf.org/doc/html/rfc9110/

Freed N. & Borenstein N. 1996. RFC 2045. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. Verkkosivu. Viitattu 23.9.2023. https://datatracker.ietf.org/doc/html/rfc2045

Fielding R. & Taylor R. 2020. Principled Design of the Modern Web Architecture. Pdf-dokumentti. Viitattu 6.2.2024. https://ics.uci.edu/~fielding/pubs/webarch_icse2000.pdf

Fielding R. 2000. Architectural Styles and the Design of Network-based Software Architectures. Verkkosivu. Viitattu 6.2.2024. https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Haro J. 2023. Microservice APIs. Manning Publications. O'Reilly Media, Inc.

Amazon Web Services. n.d. Caching Overview. Verkkosivu. Viitattu 27.2.2024. https://aws.amazon.com/caching/

Kotilainen, O. 2009. Representational State Transfer (REST) ja Web-suuntautunut arkkitehtuuri (WOA) arkkitehtuurityylinä. Luento 14.5.2009. Seminaari: Palvelusuuntautuneet järjestelmät. Julkaisija Helsingin yliopisto. Viitattu 29.2.2024. https://www.cs.helsinki.fi/group/cinco/teaching/2009/soc-seminaari/

OpenAPI Initiative. n.d. What is OpenAPI? Verkkosivu. Viitattu 5.9.2024. https://www.openapis.org/what-is-openapi

Gardiner, M. 2018. Better API Design with OpenAPI. Puheenvuoro 26.7.2018. Google Cloud Next ‘18. YouTube-video. Viitattu 13.9.2024. https://www.youtube.com/watch?v=uBs6dfUgxcI
Target, S. 2017. The Rise and Rise of JSON. Verkkosivu. Viitattu 13.9. https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html

OpenAPI Initiative. n.d. Style Guide. Verkkosivu. Viitattu 20.9.2024. https://www.openapis.org/faq/style-guide

OpenAPI Initiative. n.d. OpenAPI Initiative Regisry. Verkkosivu. Viitattu 21.9.2024. https://spec.openapis.org/

Rosenstock, K. & Ponelat J. 2022. Designing APIs with Swagger and OpenAPI. Manning Publications.

OpenAPI Initiative. 2021. OpenAPI Specification v3.1.0. Verkkosivu. Viitattu 26.9.2024. https://spec.openapis.org/oas/latest.html

OpenAPI Initiative. 2021. Migrating from OpenAPI 3.0 to 3.1.0. Verkkosivu. Viitattu 26.9.2024. https://www.openapis.org/blog/2021/02/16/migrating-from-openapi-3-0-to-3-1-0

Desrosiers J. 2021. JSON Schema. Validating OpenAPI and JSON Schema. Verkkosivu. Viitattu 11.10.2024. https://json-schema.org/blog/posts/validating-openapi-and-json-schema#how-does-it-work

YAML. 2021. YAML Ain't Markup Language (YAML™) version 1.2. Revision 1.2.2 (2021-10-01). Verkkosivu. Viitattu 11.10.2024. https://yaml.org/spec/1.2.2/

Tucci M. 2023. Swagger. Code-First vs. Design-First: Eliminate Friction with API Exploration. Verkkosivu. Viitattu 3.11.2024. https://swagger.io/blog/code-first-vs-design-first-api/

OpenAPI Initiative. n.d. Getting started. Verkkosivu. Viitattu 8.11.2024. https://learn.openapis.org/
Swagger. n.d. Swagger UI. Verkkosivu. Viitattu 8.11.2024. https://swagger.io/tools/swagger-ui/

Prism. n.d. HTTP Mocking. Verkkosivu. Viitattu 23.11.2024. https://docs.stoplight.io/docs/prism/83dbbd75532cf-http-mocking

Prism. n.d. Dynamic Response Generation with Faker. Verkkosivu. Viitattu 23.11.2024. https://docs.stoplight.io/docs/prism/9528b5a8272c0-dynamic-response-generation-with-faker

Prism. n.d. Validation Proxy. Verkkosivu. Viitattu 10.11.2024. https://docs.stoplight.io/docs/prism/72d69fb629de0-validation-proxy

OpenAPI-core. n.d. Validation. Verkkosivu. Viitattu 10.11.2024. https://openapi-core.readthedocs.io/en/latest/validation/

PayU GPO. n.d. openapi-validator-middleware. Verkkosivu. Viitattu 10.11.2024. https://www.npmjs.com/package/openapi-validator-middleware

Trudeau C. 2024. Django in Action. Manning Publications.

Django. 2024. Django appears to be a MVC framework, but you call the Controller the "view", and the View the "template". How come you don't use the standard names? Verkkosivu. Viitattu 23.11. https://docs.djangoproject.com/en/5.1/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names

Zhou, Z. 2021. What is WSGI? A Readable Explanation for Python Developers. Verkkosivu. Viitattu 23.11.2024. https://medium.com/techtofreedom/what-is-wsgi-a-readable-explanation-for-python-developers-e91beba35965

Django. 2024. URL dispatcher. Verkkosivu. Viitattu 23.11.2024. https://docs.djangoproject.com/en/5.1/topics/http/urls/

Django. 2024. Writing views. Verkkosivu. Viitattu 23.11.2024. https://docs.djangoproject.com/en/5.1/topics/http/views/

Django. 2024. Templates. Verkkosivu. Viitattu 24.11.2024. https://docs.djangoproject.com/en/5.1/topics/templates/

Django. 2024. Models. Verkkosivu. Viitattu. 24.11.2024. https://docs.djangoproject.com/en/5.1/topics/db/models/

Django. 2024. Request and response objects. Verkkosivu. Viitattu. 24.11.2024. https://docs.djangoproject.com/en/5.1/ref/request-response/

Django REST framework. n.d. Serializers. Verkkosivu. Viitattu 27.5.2025. https://www.django-rest-framework.org/api-guide/serializers/

Django REST framework. n.d. Routers. Verkkosivu. Viitattu 27.5.2025. https://www.django-rest-framework.org/api-guide/routers/

Django REST framework. n.d. Routers. Verkkosivu. Viitattu 27.5.2025. https://www.django-rest-framework.org/api-guide/viewsets/

OpenAPI Generator. 2024. Generators List. Verkkosivu. Viitattu 13.12.2024. https://openapi-generator.tech/docs/generators

Django Ninja. 2024. Motivation. Verkkosivu. Viitattu 13.12.2024. https://django-ninja.dev/

LIITTEET

Liite 1. HTTP-metodit

HTTP-metodi	**Selite**
GET	Hae resurssi
POST	Luo uusi resurssi
PUT	Korvaa olemassa oleva resurssi
PATCH	Muokkaa osaa olemassa olevasta resurssista
DELETE	Poista resurssi
HEAD	Hae resurssin otsakemetadata
OPTIONS	Mitä HTTP-metodeja palvelin tukee
TRACE	Miten HTTP-pyyntö muuttui ja mitä kautta pyyntö päätyi kohteeseen
CONNECT	Luo HTTP-tunneli asiakasohjelman ja palvelimen välille

Liite 2. HTTP-tilakoodit ja selitetekstit

Value	Description
100	Continue
101	Switching Protocols
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
306	(Unused)
307	Temporary Redirect
308	Permanent Redirect
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Content Too Large
414	URI Too Long
415	Unsupported Media Type
416	Range Not Satisfiable
417	Expectation Failed
418	(Unused)
421	Misdirected Request
422	Unprocessable Content
426	Upgrade Required
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported