Leevi Koskinen

Välityspalvelin API-versioiden hallintaan ja yhteensopivuuksien varmistamiseen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Ohjelmistotuotanto

Insinöörityö

13.10.2024

Tiivistelmä

Tekijä:	Leevi Koskinen
Otsikko:	Välityspalvelin API-versioiden hallintaan ja yhteensopivuuksien varmistamiseen
Sivumäärä:	35 sivua
Aika:	13.10.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Vesa Ollikainen

Tässä insinöörityössä suunnitellaan ja toteutetaan välityspalvelimen demo API-versioiden hallintaan ja yhteensopivuuksien varmistamiseen. Työn lähtökohtana on kuvitteellisen yrityksen ongelma, jossa vanhat sovellukset eivät ole yhteensopivia kaikkien uusien API-versioiden kanssa. Ratkaisuna kehitetään välityspalvelin, joka toimii sovellusten ja API-palveluiden välissä muuntaen ja emuloiden poistuneiden API-versioiden vastauksia.

Välityspalvelin mahdollistaa vanhojen sovellusten käytön ilman massiivisia päivitystarpeita. Työ tutkii välityspalvelimien ja API-yhdyskäytävien toiminnallisuuksia sekä eri teknologioiden kuten Dockerin, REST-rajapintojen ja .NET Coren käyttöä järjestelmän toteutuksessa. Välityspalvelinratkaisu tarjoaa joustavan tavan hallita API-päivityksiä ja ylläpitää vanhoja järjestelmiä kustannustehokkaasti.

Avainsanat: API, välityspalvelin, API-yhdyskäytävä, REST-rajapinta

---

Tämän opinnäytetyön alkuperä on tarkastettu Turnitin Originality Check -ohjelmalla.

Abstract

Author:	Leevi Koskinen
Title:	Proxy Server for Managing API Versions and Ensuring Compatibility
Number of Pages:	35 pages
Date:	13 October 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisors:	Vesa Ollikainen, Senior Lecturer

This thesis focuses on the design and implementation of a proxy server for managing API versions and ensuring compatibility. The starting point is the problem faced by a fictional company where older applications are not compatible with newer API versions. The solution involves developing a proxy server that sits between the applications and API services, transforming and emulating responses from deprecated API versions.

The proxy server allows the continued use of older applications without the need for significant updates. The thesis explores the functionalities of proxy servers and API gateways, as well as the use of technologies such as Docker, REST APIs, and .NET Core in the system implementation. The proposed proxy server solution offers a flexible approach to managing API updates while maintaining older systems in a cost-effective manner.

Keywords: API, proxy server, API gateway, REST API

Sisällys

Lyhenteet

1 Johdanto 1

2 Välityspalvelin 3

2.1 Mikä on välityspalvelin? 3
2.2 Välittävä välityspalvelin 5
2.3 Käänteinen välityspalvelin 6

3 API-yhdyskäytävä 7

3.1 Mikä on API-yhdyskäytävä? 7
3.2 API-yhdyskäytävän arkkitehtuuri 9

4 Demon suunnittelu ja toteutus 10

4.1 Suunnittelumallit 10
4.2 Järjestelmän teknologiat 11
4.2.1 Docker 12
4.2.2 REST-rajapinta 13
4.2.3 Swagger/OpenAPI 14
4.2.4 .Net Core 15
4.3 Järjestelmän toteutus 16
4.3.1 Projektin aloitus 17
4.3.2 Datan mallinnus 20
4.3.3 Datan muuntaminen 22
4.3.4 Rajapintojen kutsuminen 28

5 Yhteenveto 33

Lähteet 34

Lyhenteet

API: *Application Programming Interface*. Ohjelmointirajapinta, joka mahdollistaa kahden eri järjestelmän toiminnan yhdessä toistensa kanssa.

CIL: *Common Intermediate Language*. CIL on keskitason kieli, joka sisältää binäärikäskyjä.

HTTP: *Hypertext Transfer Protocol*. Hypertekstin siirtoprotokolla on vuonna 1991 julkaistu tilaton protokolla, jonka avulla internetissä voidaan jakaa resursseja, kuten verkkosivuja ja tiedostoja.

IP: *Internet Protocol*. Yksilöivä osoite, jonka perusteella laite tunnistetaan Internetissä tai paikallisessa verkossa.

JSON: *JavaScript Object Notation*. JSON on merkintäkieli, joka on yksinkertainen ja kevyt avoimen standardin tiedostomuoto tiedonvälitykseen ja tallennukseen.

REST: *Representational State transfer*. HTTP-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen.

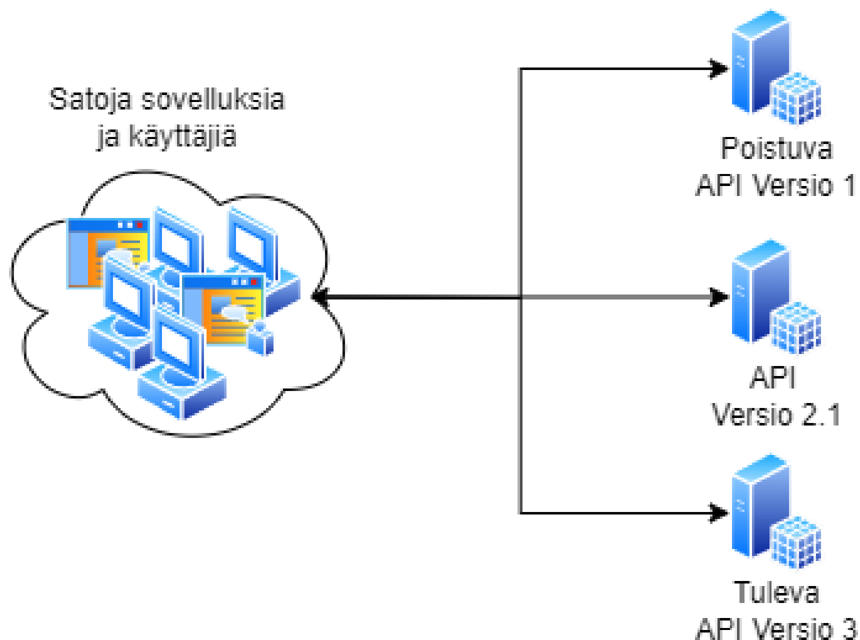
VPN: *Virtual Private Network*. Erillisverkko, jonka tarkoituksena on salata yhteydet sen kautta internetiin ja piilottaa alkuperäinen IP-osoite.

XML: *Extensible Markup Language*. XML on merkintäkieli, jota käytetään sekä tiedonvälitykseen järjestelmien välillä että tiedostomuotona dokumenttien tallentamiseen.

1 Johdanto

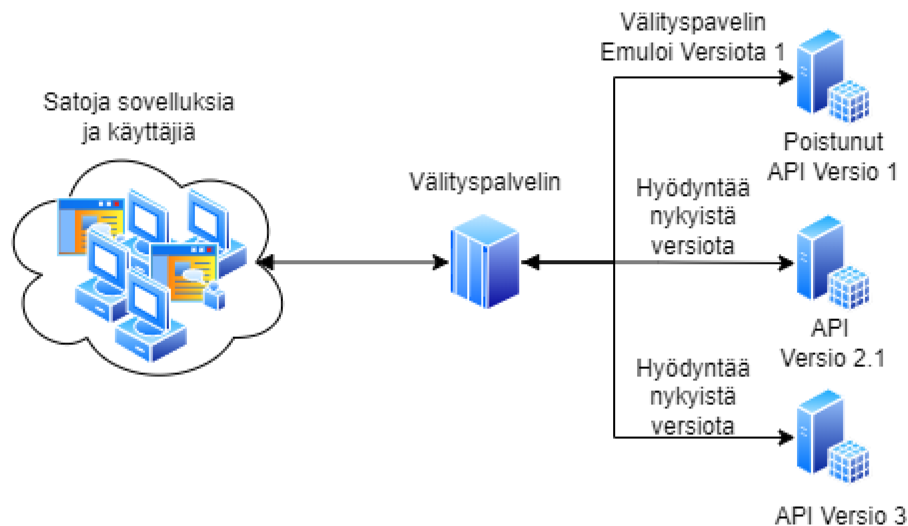
Tässä insinöörityössä tulemme suunnittelemaan ja rakentamaan demovälityspalvelimen API (application programming interface) sovellusrajapintaversioiden hallintaan ja yhteensopivuuksien varmistamiseen. Käymme yleisellä tasolla läpi, mitä ovat välityspalvelimet ovat ja mitä hyötyjä ne tuovat. Tutustumme myös API-yhdyskäytävään, joka on ohjelmistomalli ja välityspalvelimen muoto. Tutkimme eri teknologioita ja ohjelmistotekniikoita, joita tulemme käyttämään demosovellukseen.

Insinöörityön aihe perustuu oikeaan ongelmaan, johon kuvitteellinen yritys on törmännyt. Yrityksellä on useita API-palveluita, joihin on liitetty paljon vanhoja sovelluksia ja käyttäjiä. Nyt nämä API-palvelut ovat päivittymässä, ja vanhat sovellukset eivät tule toimimaan suoraan uusien versioiden kanssa. Yksinkertaistetusta kuvasta 1 pystymme näkemään yrityksen nykytilanteen, jossa useat ohjelmistot ja käyttäjät ovat suorassa yhteydessä API-palveluihin.



Kuva 1. Nykytilanne ja yrityksen ongelma yksinkertaistettuna.

Kuten arvata saattaa, tämä nykyinen tilanne tulee aiheuttamaan suuren ongelman yritykselle, koska vanhojen ohjelmistojen päivittäminen ei ole helppoa ja halpaa. Yritys tulee kuluttamaan paljon resursseja ja aikaa päivittämällä satoja ohjelmistoja ja järjestelmiä. Tätä varten lähdemme tässä insinöörityössä kehittämään välityspalvelinta, joka sijaitsee API-palveluiden ja sovellusten välissä. Kuvasta 2 pystymme näkemään, mihin väliin olemme rakentamassa välityspalvelintamme. Palvelin tulee sovellusten ja API-palveluiden väliin.



Kuva 2. Yrityksen ongelman ratkaisu.

Kuvasta 2 myös pystymme päättelemään, että välityspalvelimen tehtävänä on tunnistaa API-kutsuja, muuntaa ja emuloida poistuneiden rajapintojen versio vastauksia riippuen, mitä rajapintaversiota sovellus haluaa. Jos esimerkiksi sovellus haluaisi poistuneen API-version 1 dataa, joutuu välityspalvelin hakemaan datan muista API-versioista ja palauttamaan datan API-version 1 muodossa. Tällaisen välityspalvelimen avulla yritys pystyy keskittymään kasvuun ja päivittämään vanhoja sovelluksia sitä mukaan, mitä pystyy, eikä kerralla kaikkia.

2 Välityspalvelin

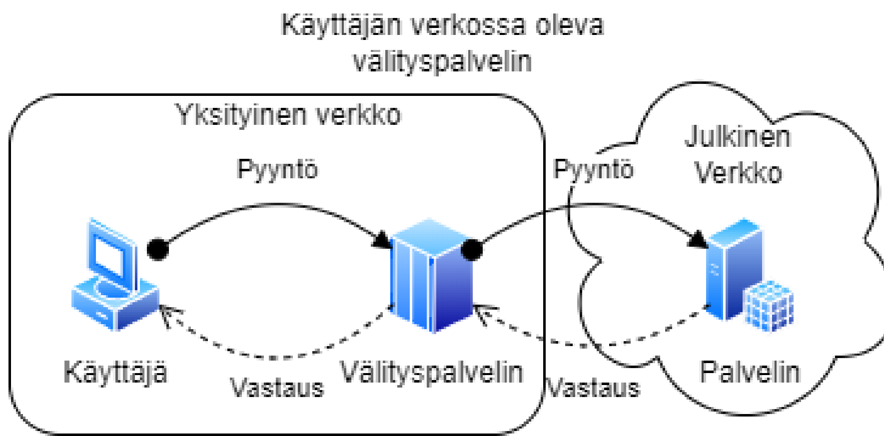
Internetissä on paljon erilaisia palvelimia ja laitteita, joista tavalliset käyttäjät eivät ole edes tietoisia. Tulevissa alaluvuissa käymme läpi yhden internetin tärkeimmistä järjestelmistä, joka on nimeltään välityspalvelin. Vastaamme kysymyksiin, mitä ne ovat ja mitä ne tekevät. Tutustumme esimerkkeinä myös tuttuihin palveluihin, jotka hyödyntävät välityspalvelimia.

2.1 Mikä on välityspalvelin?

Välityspalvelin on tavallisesti tietokonejärjestelmä, joka on käyttäjän ja palvelinjärjestelmän verkkoyhteyden välissä. Välityspalvelimen perusperiaate on vastaanottaa käyttäjän pyyntöjä ja välittää ne eteenpäin halutulle palvelimelle. Välityspalvelin palauttaa käyttäjälle kohdepalvelimen vastauksen alkuperäisessä muodossa tai muunnetussa muodossa riippuen, miten välityspalvelin on suunniteltu ja toteutettu. [1.]

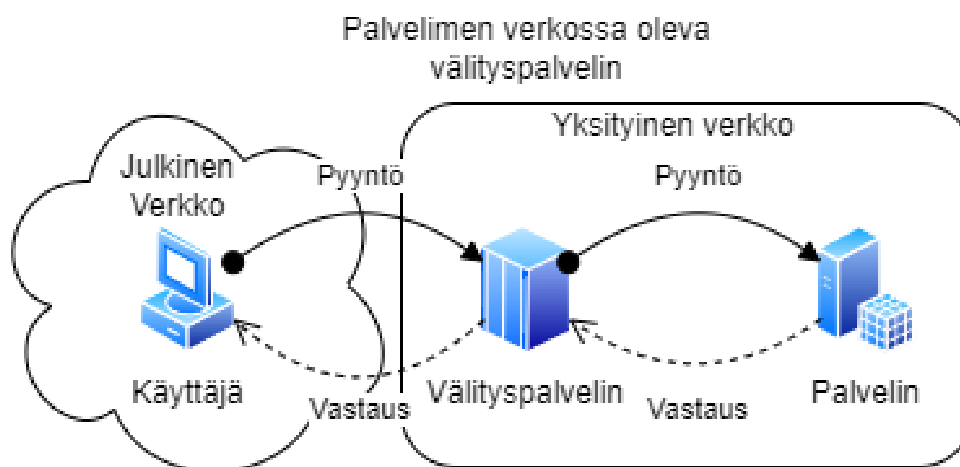
Pääsääntöisesti välityspalvelimet pystytään jakamaan kahteen pääkategoriaan, jotka ovat englanniksi nimellä ”forward proxy” ja ”reverse proxy” [2]. ”Forward proxy”:n tulen kääntämään suomeksi vapaasti muotoon välittävä välityspalvelin, koska sillä ei ole olemassa vakiintunutta suomennosta, muuta kuin välityspalvelin. Näin pystymme erottamaan selkeästi, kun puhutaan yleisesti välityspalvelimista tai välittävästä välityspalvelimesta. ”Reverse proxy”:lla on valmiiksi vakiintunut käännös, joka on käänteinen välityspalvelin.

Kuvissa 3 ja 4 pystymme huomaamaan kuinka, nämä kaksi pääkategoriaa eroavat toisistaan. Kuvassa 3 välittävä välityspalvelin toimii käyttäjän omassa yksityisessä verkossa. Tämä välityspalvelin lähettää käyttäjän puolesta tiedot palvelimelle ja palauttaa palvelimen vastauksen. [2.]



Kuva 3. Välittävä välityspalvelimen toiminta verkossa.

Kuvassa 4 käänteinen välityspalvelin toimii palvelimen yksityisessä verkossa, jolloin se suojaa varsinaisesti palvelinta eikä käyttäjää. Välityspalvelin lähettää käyttäjän pyynnön kohdepalvelimelle ja palauttaa palvelimen vastauksen takaisin käyttäjälle. [2.]

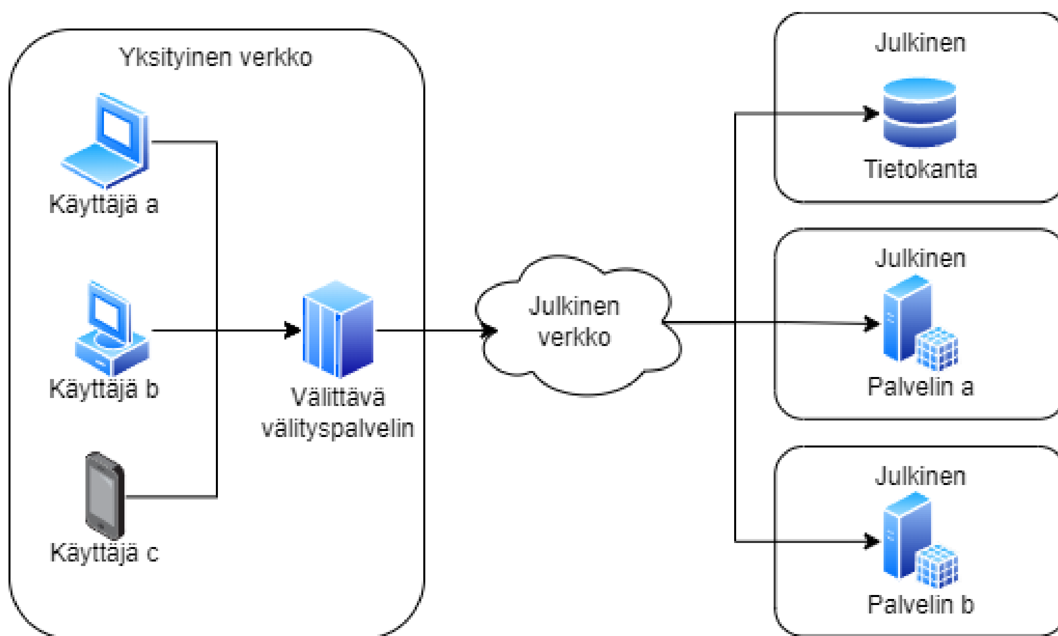


Kuva 4. Käänteinen välityspalvelin toiminta verkossa.

Näiden kahden pääkategorian ero on huomattava etenkin tietoturvallisuudessa. Esimerkiksi välittävä välityspalvelin suojaa käyttäjän Internetin protokollaosoitteen eli IP-osoitteen. Käänteinen välityspalvelin suojaa palvelimen hyväksymällä vain haluttuja kutsuja tai käyttäjiä. Käymme tulevissa alaluvuissa tarkemmin läpi nämä pääkategoriat. [2.]

2.2 Välittävä välityspalvelin

Välittävä välityspalvelin on käyttäjän yksityisverkon sisällä oleva palvelin, joka ohjaa julkiseen verkkoon meneviä kutsuja. Kuvasta 5 pystymme toteamaan, kuinka käyttäjät A, B ja C ovat yhteydessä välittävään välityspalvelimeen ennen kuin pääsevät julkiseen verkkoon. Tämä välityspalvelin suojaa näitä käyttäjiä piilottamalla niiden IP-osoitteet ja hallitsee, minne kaikkialle käyttäjillä on oikeus lähettää kutsuja. Esimerkiksi jos haluamme estää opiskelijoita pelaamasta koulun verkossa uhkapelejä, pystymme välittävän välityspalvelimen kanssa estämään veikkaus.fi-palvelimen kokonaan. [3.]



Kuva 5. Välityspalvelin käyttäjien suojana.

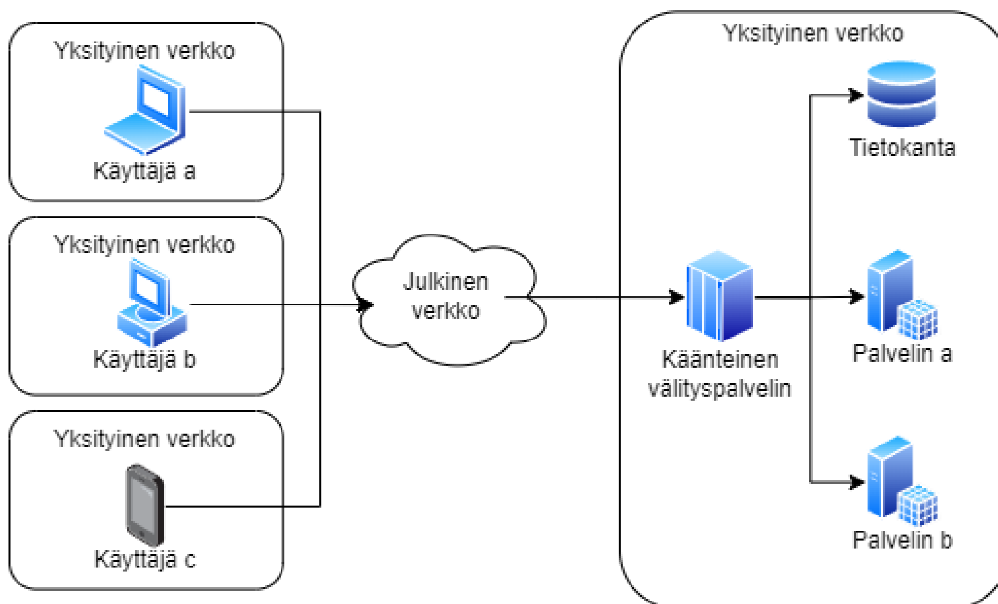
Välittävä välityspalvelin myös pystyy taltioimaan välimuistiin eli cahceen usein pyydettyjä pyyntöjä ja niiden vastauksia. Tällä keinolla välityspalvelin pystyy tarjoamaan vastauksia erittäin nopeasti ja mahdollisesti nopeammin kuin kohdepalvelin. Tämän toiminnan avulla välityspalvelimet pystyvät vähentämään kohdepalvelimen kuormitusta, verkkokaistan kulutusta ja käyttöastetta. [4.]

Välittävä välityspalvelin myös pystyy sijaitsemaan julkisessa verkossa ja toimimaan ”Virtual Private Network” eli VPN:n tavoin. VPN on siis julkinen

välityspalvelin. Tällä käyttäjät pystyvät esimerkiksi suojaamaan alkuperäisen verkkonsa tai kiertämään maanosarajoituksia. [4.]

2.3 Käänteinen välityspalvelin

Käänteinen välityspalvelin on samanlainen kuin välittävä välityspalvelin, mutta sen sijaan, että se suojaa käyttäjiä se, suojaa palvelimia. Käänteinen välityspalvelin suodattaa ja hyväksyy käyttäjien kutsuja palvelimelle. Käyttäjät eivät näe välityspalvelimen takana tapahtuvista asioista, jolloin prosessoinnin pystyy hoitamaan mikä tahansa palvelin, jonka välityspalvelin määrittää. [4.] Kuvasta 6 näemme miten, eri käyttäjät yhdistyvät internetin kautta ensin käänteiseen välityspalvelimeen ja käänteinen välityspalvelin ohjaa prosessoiviin palvelimiin.



Kuva 6. Välityspalvelin palvelimien suojana.

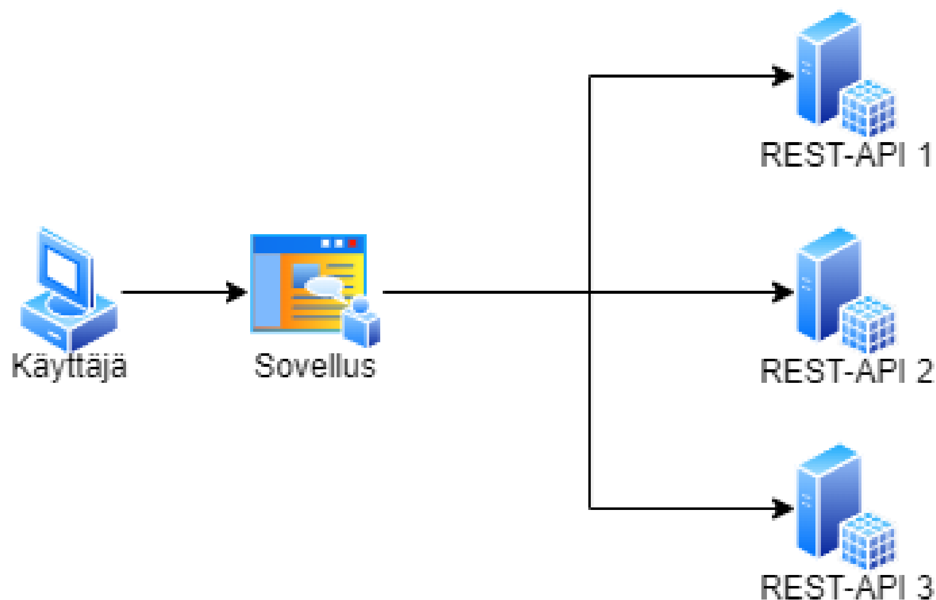
Käänteinen välityspalvelin pystyy tasapainottamaan palvelimien kuormitusta ohjaamalla pyyntöjä eri palvelimille. Näin toimimalla pystytään parantamaan vasteaikoja ja vähentämään palveluiden kuormittumista ja hidastumista. Käänteiset välityspalvelimet pystyvät myös välittävän välityspalvelimen kaltaisesti tallentamaan välimuistiin usein toistuvia kutsuja ja niiden vastauksia. Näin välityspalvelimet pystyvät vähentämään verkkoliikennettä ja rasitusta palvelimille. [5.]

3 API-yhdyskäytävä

3.1 Mikä on API-yhdyskäytävä?

API-yhdyskäytävä on ohjelmistomalli, jonka tehtävänä on toimia yksittäisenä rajapintana, joko ohjelmisto rajapinnan edessä tai useamman mikropalvelun pyyntöjen ja tietojen välittäjänä. API-yhdyskäytävän avulla pystytään standardisoimaan organisaation ohjelmistojen, tiedon ja palvelujen välisiä toimintoja. API-yhdyskäytävän avulla organisaatio pystyy jakamaan dataa organisaation sisällä ja sen ulkopuolelle hallitun yksittäisen rajapinnan avulla. [6.]

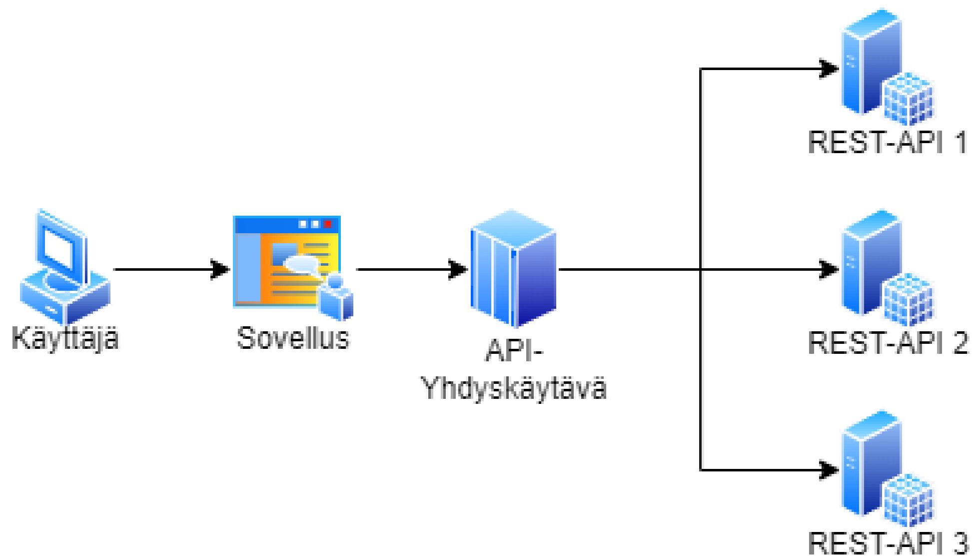
Yksittäisen rajapinnan avulla myös pystytään helpommin ylläpitämään ja käyttämään isoja järjestelmiä, joissa on monta yksittäistä pientä mikropalvelua [6]. Kuva 7 kuvastaa, kuinka yksi sovellus on yhdistetty useampaan REST-rajapintaan, jotka toimivat mikropalveluina.



Kuva 7. Sovellus yhdistää suoraan mikropalveluihin.

Tällaista kuvan 7 esittämää sovellusta on hankalampi ylläpitää ja kehittää, koska jokainen yhteys ja rajapintakutsun toimivuus pitää varmistaa ja testata. Useamman rajapinnan käyttäminen myös tekee ohjelmistosta heti monimutkaisemman.

Kuva 8 kuvastaa, kuinka yhden API-yhdyskäytävän taakse on laitettu useampi mikropalveluja kuvaava REST-rajapinta. API-yhdyskäytävä pystyy kommunikoimaan jokaisen REST-rajapinnan kanssa ja yhdistämään useamman rajapinnan vastaukset kerralla yhdeksi vastaukseksi.



Kuva 8. Sovellus yhdistää API-yhdyskäytävällä mikropalveluihin.

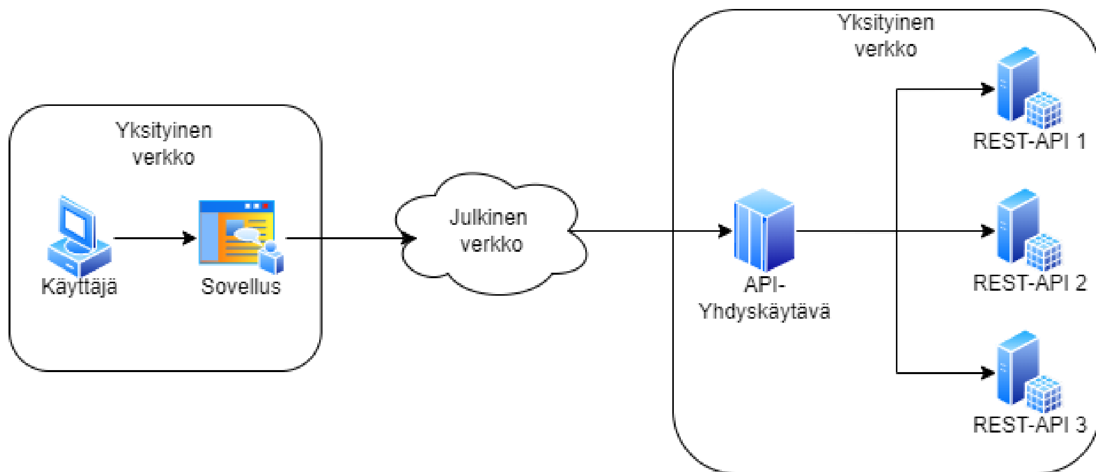
API-yhdyskäytävän päätehtävänä on siis toimia yksittäisenä rajapintana, joka varmistaa kutsun todennuksen eli onko sovelluksella tai käyttäjällä oikeus tehdä kutsu. Lisäksi API-yhdyskäytävä tasapainottaa kutsut eri palvelimille eli varmistaa, ettei yksittäinen palvelin kuormitu enemmän kuin muut. API-yhdyskäytävä myös pystyy muuntamaan kutsujen protokollia palveluille sopiviksi. Lisäksi se seuraa, kirjaa ja analysoi kutsuja ja virheilmoituksia. [6; 7.]

Näin ollen pystymme toteamaan, että API-yhdyskäytävät ovat erittäin hyödyllisiä, kun sovellus tai järjestelmä käyttää useampaa mikropalvelua. Esimerkiksi tunnettu Uber käyttää API-yhdyskäytävää yhdistääkseen kriittisiä mikropalveluja. Uberille näitä mikropalveluja on kertynyt jo 2200 kappaletta, ja niiden kaikkien ylläpitäminen, seuranta, tietoturva ja toimivuuden varmistaminen aiheuttavat jo

suuria haasteita. API-yhdyskäytävän avulla he pystyivät luomaan useamman tiimin, jotka kehittävät ja ylläpitävät omia mikropalveluja ja koko järjestelmästä tuli helpommin ylläpidettävä. [8.]

3.2 API-yhdyskäytävän arkkitehtuuri

Nyt kun tiedämme pääpiirteittäin, mikä API-yhdyskäytävä on, voimme lähteä syventymään sen arkkitehtuuriin. Luvun 2 avulla pystymme päättelemään, että API-yhdyskäytävä on käänteinen välityspalvelin, ja tätä päätelmää tukee myös Mehmet Ozkayan medium artikkelissaan, jossa hän myös toteaa, että API-yhdyskäytävä on käänteinen välityspalvelin. [9.] Kuvasta 9 huomaamme, kuinka API-yhdyskäytävä sijaitsee mikropalvelujen edessä.



Kuva 9. API-yhdyskäytävä.

API-yhdyskäytävän suunnittelussa pitää huomioida, ettei vahingossa luoda järjestelmään yksittäistä heikkoa kohtaa. Jos esimerkiksi luodaan vain yksi API-yhdyskäytävä, joka sijaitsee vain yhdellä palvelimella, aiheutamme ison riskin, että koko järjestelmä kaatuu, kun API-yhdyskäytävä kaatuu. Näin ollen on kannattavaa jakaa API-yhdyskäytävä useampaan osaan, joista kukin hoitaa osan mikropalveluista. Esimerkiksi yrityslogiikka ja puhdas ohjelmistologiikka kannattaa olla omissa API-yhdyskäytävissään. [9.]

4 Demon suunnittelu ja toteutus

Edellisissä luvuissa kävimme läpi, mitä ovat välityspalvelimet ja API-yhdyskäytävät. Näin ollen pystymme nyt käsittämään, mitä järjestelmää olemme rakentamassa. Demo tulee olemaan yksinkertainen kirjakauppa rajapinta, jonka tehtävänä on yhdistää vanhat kirjarajapintojen tietojärjestelmät yhteen. Demon tehtävänä on palauttaa, lisätä ja muokata kirjojen tiedot halutussa datamuodossa.

Ennen kuin pääsemme itse demon rajapinnan eli API-yhdyskäytävän ohjelmointiin, kannattaa meidän lähteä tutustumaan käytettäviin teknologioihin. Demon suunnittelussa tulemme hyödyntämään suunnittelumalleja, jotta pystymme luomaan selkeät toimintamallit. Suunnittelumallit helpottavat tulevaisuudessa päivittämään API-yhdyskäytäväämme.

4.1 Suunnittelumallit

Olio-ohjelmoinnissa ohjelman suunnittelu on haasteellista, ja etenkin uudelleen käytettävän koodin suunnittelu tuottaa haasteita niin uusille kehittäjille kuin vanhoille senior-tason kehittäjille. Luokan, rajapinnan tai perintähierarkian tulee olla suunniteltu niin, että tämänhetkinen ongelma saadaan kuvattua ja ratkaistua. Samaan aikaa suunniteltu koodi pitäisi olla mahdollisimman yleispätevä, jotta tulevaisuudessa tulevat muutokset ja vaatimukset on otettu jo huomioon ja jo kirjoitettua koodia käytettäisiin uudelleen. [10.]

Koodin uudelleenkäytettävyyden avuksi on luotu suunnittelumalleja, joilla pystytään ratkaisemaan näitä ohjelmiston suunnittelussa toistuvia haasteita ja ongelmia. Usein ohjelmoijat törmäävät tilanteisiin, joissa joitakin suunnittelumallia olisi hyvä soveltaa. Esimerkiksi pieni luokkamuutos saattaa aiheuttaa suuremman ohjelmistoremontin, kun kyseistä luokkaa kutsutaan ja käytetään joka paikassa.

Yksinkertaistettuna suunnittelumallit auttavat ohjelmistojen suunnittelua, parantavat rakenteen ymmärrettävyyttä, selkeyttävät koodia ja lisäävät koodin uudelleenkäytettävyyttä.

Suunnittelumallit voidaan luokitella tarkoituksien mukaan eli sen mukaan millaista, ongelmaa ne ratkaisevat. Taulukosta 1 pystymme näkemään suunnittelumallien luokituksia, joita Gamma on kirjassaan esitellyt. [10.]

Taulukko 1.Gamman suunnittelumallit.

		Tarkoitus:		
		Luontimallit	Rakennemallit	Käyttäystymismallit
Kohde:	Luokka	Tehdasfunktio	Sovitin	Tulkki Yleisfunktio
	Olio	Abstrakti tehdas Rakentajarajapinta Prototyyppi Ainokainen	Sovitin Silta Rekursiokooste Kuorruttaja Julkisivu Hiutale Edustaja	Vastuuketju Komento Iteraatio Välittäjä Muisto Tarkkailija Tila Strategia Vierailija

Suunnittelumalleilla on neljä tärkeää osaa

- mallin nimi
- ongelma
- ratkaisu
- seuraukset.

Nimen tarkoitus on olla kuvaava, josta saa mahdollisimman helposti ongelman kuvauksen, ratkaisun ja seuraukset ilmi. Ongelma kuvaa, milloin suunnittelumallia kannattaa käyttää. Ratkaisu kuvaa suunnittelumallin toimintaa, suhteita ja vastuita abstraktilla tasolla, eikä se kuvaa itse toteutusta. Seuraukset kuvaavat etuja, haittoja ja kompromisseja, joita suunnittelumallin käyttäytymismallit tuovat. [10.]

4.2 Järjestelmän teknologiat

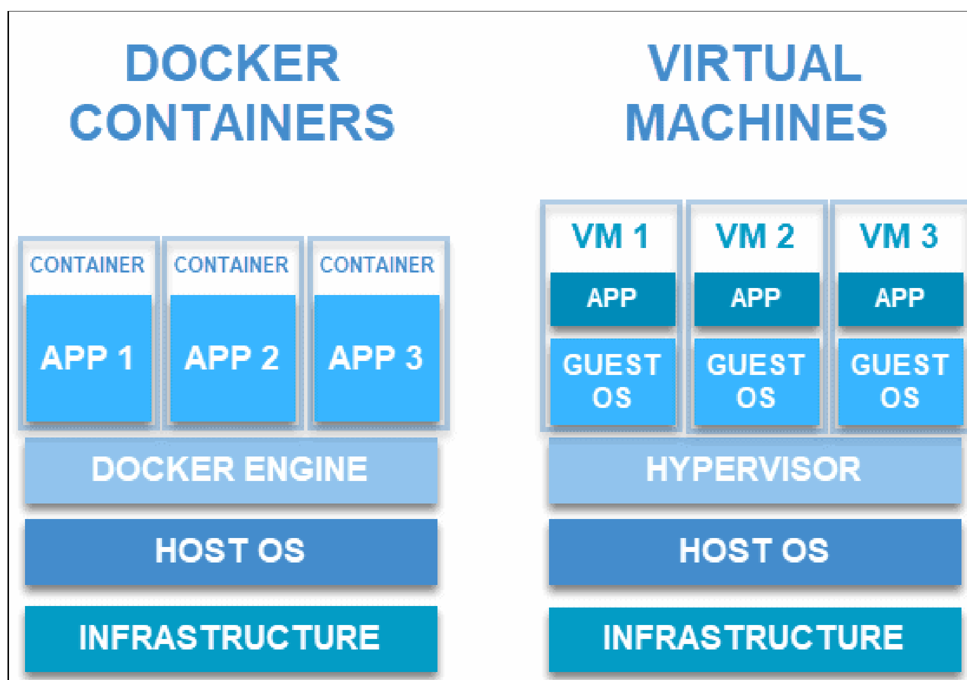
Seuraavissa alaluvuissa käymme läpi teknologioita, joita tulemme käyttämään demon kehittämisessä. Docker, .NET Core, REST-rajapinta ja Swagger

valikoituivat, koska teemme oletuksen, että kuvitteellinen yritys haluaa hyödyntää näitä teknologioita. Oletamme, että yleisinä teknologioina nämä teknologiat ovat yritykselle ja työntekijöille tuttuja ja toimivia.

4.2.1 Docker

Demon kehityksessä ja käyttöönotossa tulemme käyttämään Dockeria. Docker on ilmainen avoimen lähteen ohjelmistoalusta, joka tulee auttamaan meitä rakentamaan, kehittämään, testaamaan ja helpottamaan ohjelmaamme käyttöönottoa. Docker käyttää käyttöjärjestelmätason virtualisointiteknologiaa, mikä mahdollistaa säiliöiden (container) luonnin. [11.]

Kontit ovat vakiomuotoisia suoritettavia paketteja, jotka sisältävät kaiken, mitä ohjelmamme vaatii suorituksen ajaksi, kuten koodin, dynaamiset kirjastot ja asetukset. Nämä säiliöt paketit ovat käytännössä pieniä virtuaalikoneita ilman kokonaista käyttöjärjestelmää ja sen tuomaa raskautta. Sen sijaan ne perustuvat suoraan Linux-käyttöjärjestelmäytimeen [12]. Säiliöillä ja virtuaalikoneilla pystymme erottamaan koko ohjelmat irti pääkäyttöjärjestelmästä. Tämä tuo vakautta ja lisätietoturvasuojaa. Kuvasta 10 pystymme näkemään, kuinka Dockerin säiliöt eivät tarvitse kokonaista käyttöjärjestelmää ja kuinka käyttöjärjestelmät sijaitsevat säiliöiden tilalla, jos niitä käyttäisi. [13.]

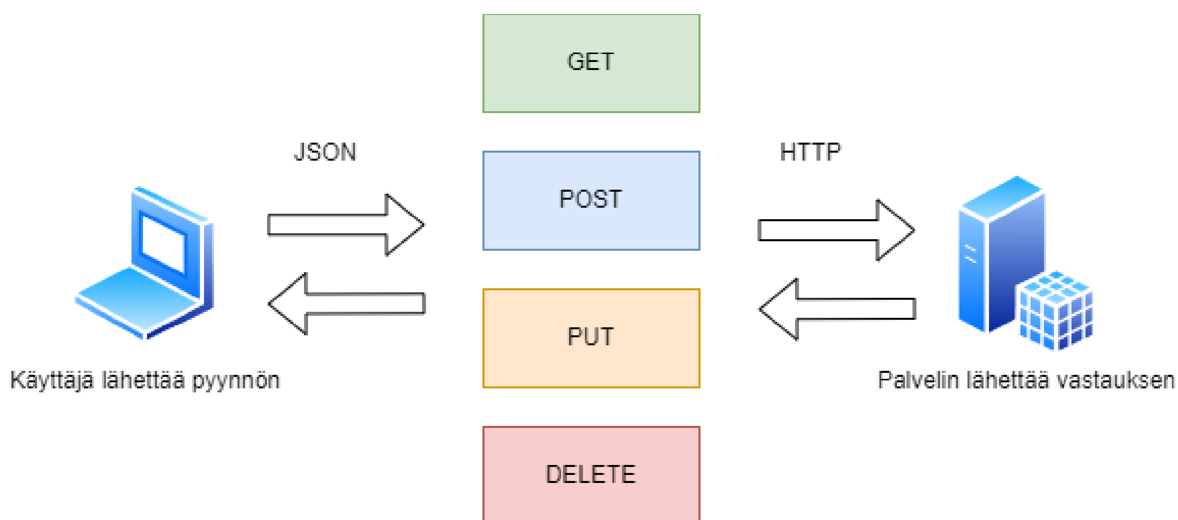


Kuva 10. Docker-säiliöt vastaan kokonaiset virtuaalikoneet.

Dockerin ja säiliöiden avulla pystymme käyttöönottamaan ohjelmamme millä tahansa alustalla hyvin helposti. Tämä perustuu siihen, että pystymme luomaan identtisen ajoympäristön riippumatta käyttöjärjestelmästä tai laitteistosta. [13.]

4.2.2 REST-rajapinta

REST-rajapinta tulee sanoista ”*Representational State Transfer*”, joka viittaa arkkitehtuurilliseen tyyliin, jonka Roy Fielding esitteli väitöskirjassaan Architectural Styles and the Design of Network-based Software vuonna 2000. [14.] REST-rajapinta kuvastaa resursseja ja operaatioita, mitä asiakas ohjelma pystyy hakemaan ja kyselemään. REST-rajapinnan ideana on yksinkertaistaa palvelinpuolen tarjoamia palveluita ja rajapintoja. REST-rajapinta hyödyntää http-protokollan metodeja ja datan mallinnuksessa XML-, JSON- tai YAML-kuvauskieltä. [15.] Kuvasta 11 näemme yksinkertaisen esityksen REST-rajapinnan toiminnasta.



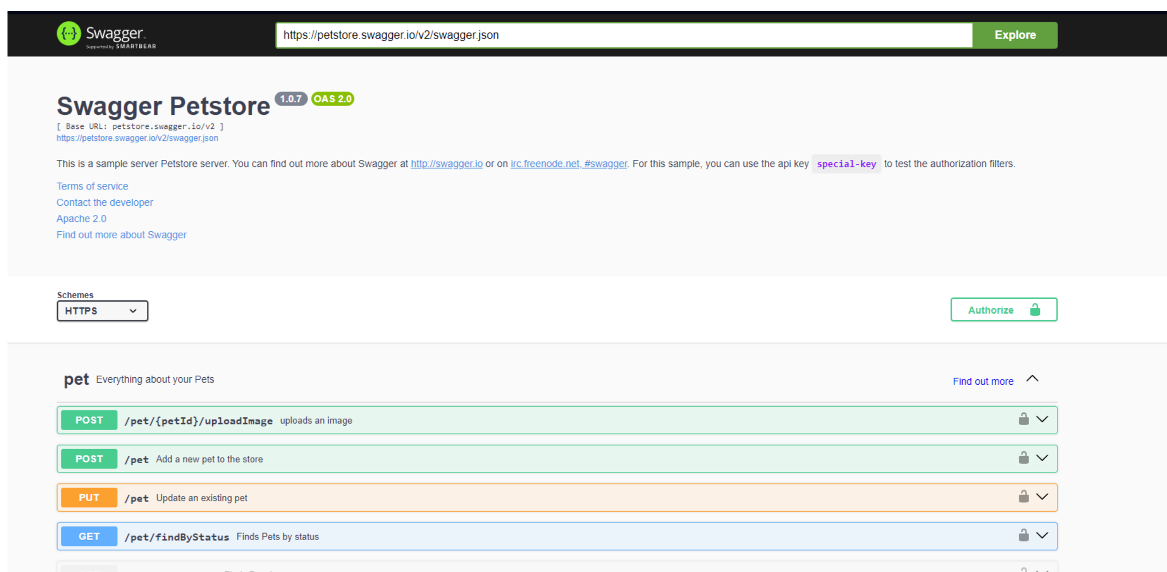
Kuva 11. REST-rajapinnan pyyntöjen toiminta.

REST-rajapinnan avulla pystymme poistamaan asiakas ohjelmien suorat yhteydet haavoittuvista järjestelmistä kuten tietokannoista. Tämä on erittäin tärkeää tietoturvallisuuden kannalta.

REST-rajapintoja dokumentoidaan hyvin usein Swagger-dokumentilla. Swagger-dokumentti pitää sisällään listan kuvauksia mitä resursseja, operaatioita ja mitä parametrejä REST-rajapinta tarjoaa ja ottaa vastaan. [15.] Käymme alaluvussa 4.2.3 läpi Swaggeria ja OpenAPI-spesifikaatiota

4.2.3 Swagger/OpenAPI

Demorajapintaa dokumentoitaessa tulemme hyödyntämään Swagger- ja OpenAPI-spesifikaatiostandardia. Swagger on REST-rajapintojen dokumentaatiota varten tehty framework ja sisältää useita hyödyllisiä työkaluja, kuten Swagger Editor, Swagger UI ja Swagger Codegen. [16.] Nämä työkalut mahdollistavat helpon dokumentoinnin ja visualisoinnin rajapinnan toiminnoista. Kuvasta 12 pysytymme näkemään, miltä dokumentoitu REST-rajapinta voisi Swagger UI:ssa näyttää.



Kuva 12. Swagger UI:n esimerkki näkymä lemmikkikaupasta.

OpenAPI-spesifikaatio määrittelee http-rajapintoja varten standardin, jota niin ihmiset kuin tietokoneetkin voivat ymmärtää. Standardin avulla ihmiset ja koneet pystyvät ymmärtämään palvelun tarjoamia toimintoja ilman pääsyä suoraan lähde koodiin. [17.]

4.2.4 .Net Core

Ohjelmistokielenä demon kehityksessä tulemme käyttämään C# ja .NET corea, koska oletamme, että kuvitteellisella yrityksellä on enemmän kokemusta C#:sta ja .NET Frameworkistä ja halua mahdollistaa ohjelmiston julkaisu myös Linux-alustalle. Framework ja Core eroavat toisistaan hyvin vähän ominaisuuksiltaan: .NET core on avoimen lähdekoodin alusta ja toimii usealla eri järjestelmällä toisin kuin .NET Framework, joka toimii ainoastaan Windows-pohjaisilla järjestelmillä. Core myös tukee muutamia ohjelmistokieliä ja useita kirjastoja. [18.]

.NET core myös tukee useita eri kohde alustoja kuten puhelimia, web-teknologioita, IoT-laitteita, pelikonsoleita ja monia muitakin niin kuin kuvasta 13 pystymme näkemään.

Kuva 13. .NET-alustoja ja työkaluja.

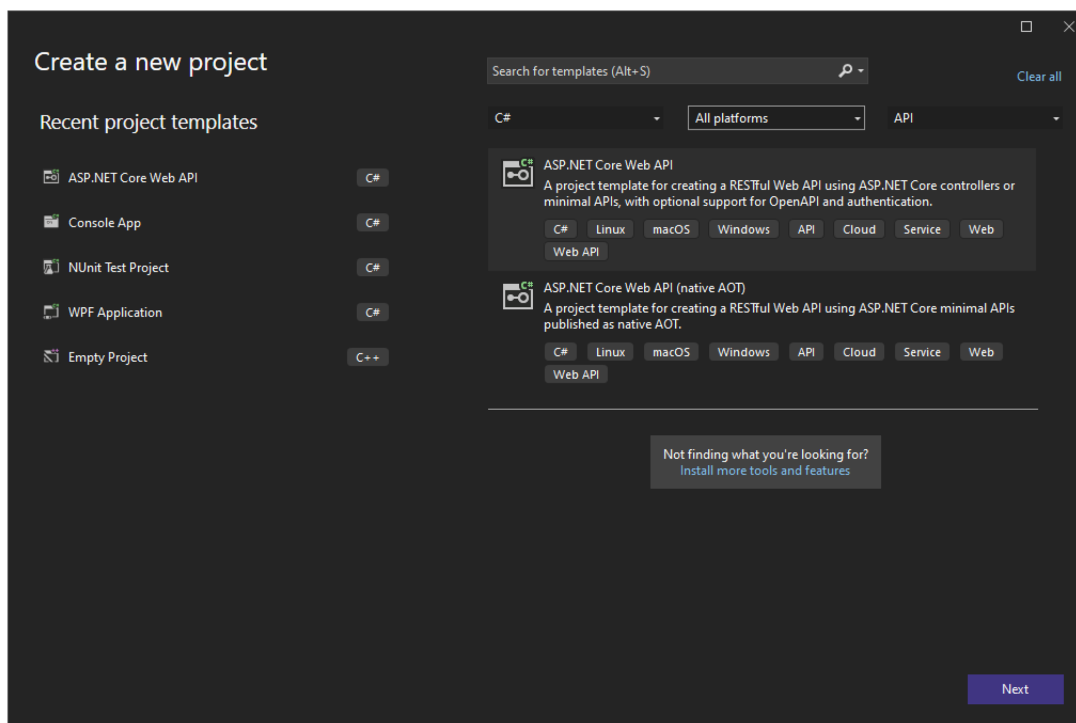
.Net Core hyödyntää Common Language infrastructurea eli CLI:tä. Siinä lähdekoodia ei käännetä binäärimuotoon vaan Common Intermediate Language eli CIL-muotoon. Tämä mahdollistaa useamman ohjelmistokielen tuen .Net coressa. [18.]

4.3 Järjestelmän toteutus

Toteutuksessa tulemme käyttämään Visual Studio 2022 Community IDE:tä. Visual Studio on Microsoftin kehittämä ohjelmointialusta, jossa on vahva .Net core ja C# tuki [19]. Kehityksen aikana tulemme käyttämään kolmea valmista kirjakauppa-API:a, joista jokaisesta saamme erilaista dataa JSON-muodossa, ja tarkoitus on säilyttää data samanlaisena riippumatta, mistä API-versiosta haemme datan.

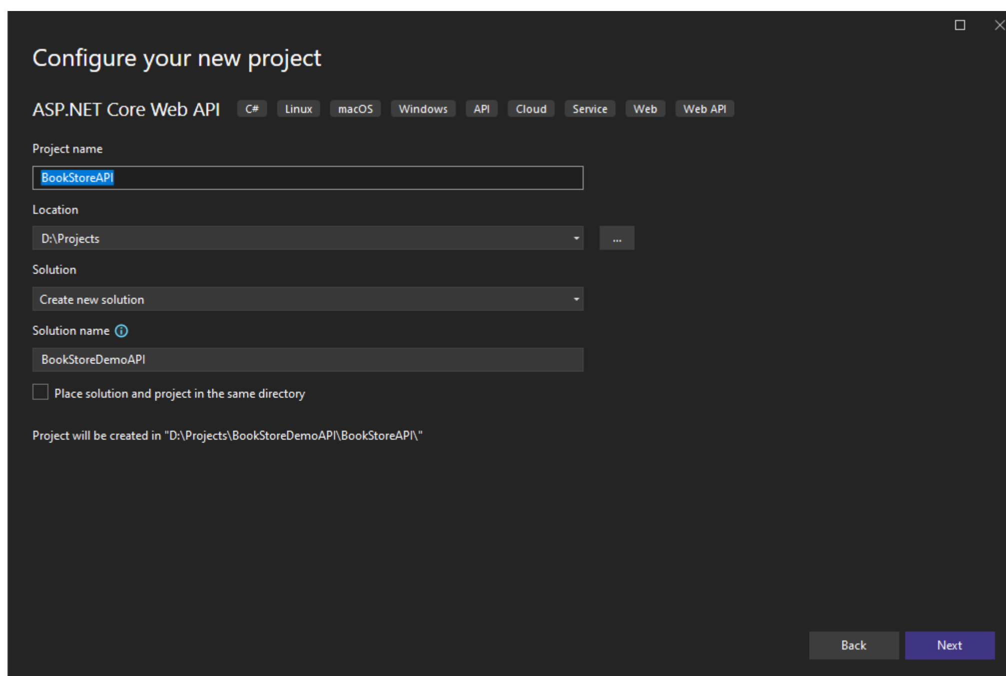
4.3.1 Projektin aloitus

Aloitamme järjestelmän luonnin tekemällä uuden projektin Visual Studion projektiluontityökalulla. Valitsemme kuvan 14 mukaisesti ASP.NET Core Web API-projektin mallipohjan, jossa projekti on valmiiksi säädetty API-kehitystä varten.



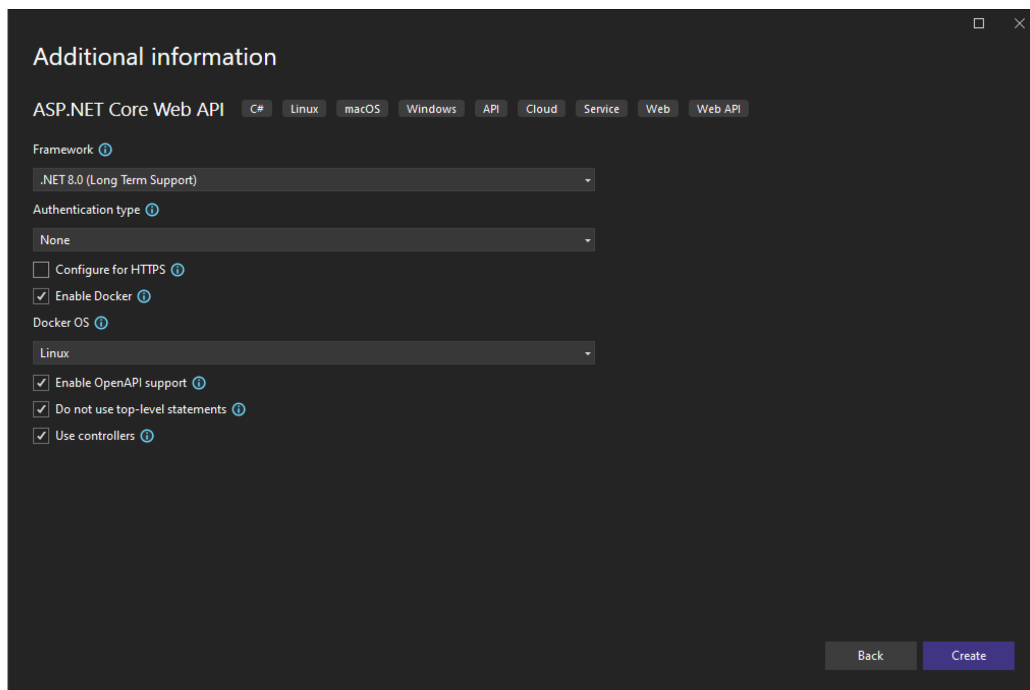
Kuva 14. Uuden projektin aloitusikkuna.

Seuraavaksi kuvan 15 mukaan lisäämme projektin perustiedot kuten nimen sekä valitsemme projektin tiedostoille paikan ja Visual Studion solution-nimen. Visual Studion solution on tekstipohjainen tiedosto, joka sisältää kaikki projektien tiedot. Tämän avulla Visual Studio pystyy hallitsemaan useita projekteja ja rakentamaan nämä projektit. [20.] Nimeämme projektin BookStoreAPI:ksi, tiedostojen paikaksi tulee D-levyn kansio Projects ja luomme Visual studio solution nimellä BookStoreDemoAPI.



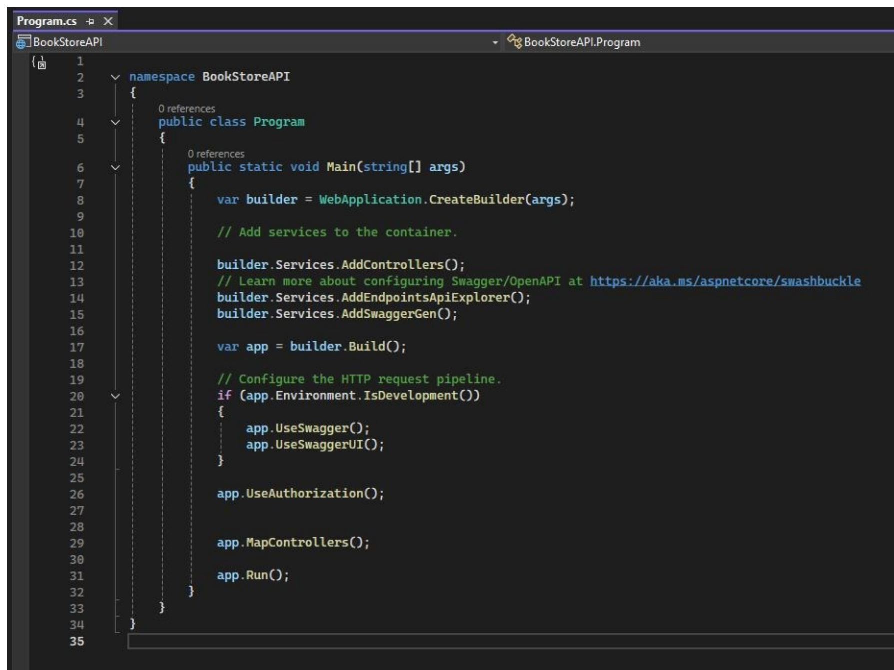
Kuva 15. Projektin perustiedot.

Seuraavaksi valitsemme kuvan 16 mukaan projektin lisätietoihin .Net 8.0-version, tunnistautumisen tyypin ei miksikään, Dockerin, OpenAPIn ja kontrollerit käyttöön. Emme valitse Top-level statement eli ylätason lauseketta, joka on C#-ominaisuus, jonka avulla voitaisiin kirjoittaa C#-ohjelman aloituspiste ilman luokkaa tai Main-metodia. Tämä yksinkertaistaa ja nopeuttaa pienien ohjelmien ja skriptien kehittämistä.



Kuva 16. Projektin lisätiedot.

Visual Studion pitäisi aueta kuvan 17 näköisesti ilman ylätason lauseketta. Program.cs sisältää näin ollen nimiavaruuden BookStoreAPI, Program-luokan ja staattisen metodin Main argumettien kanssa. Main-metodi sisältää myös valmiiksi swaggerin palvelunluontimetodit ja kontrollereiden kartoituksen.



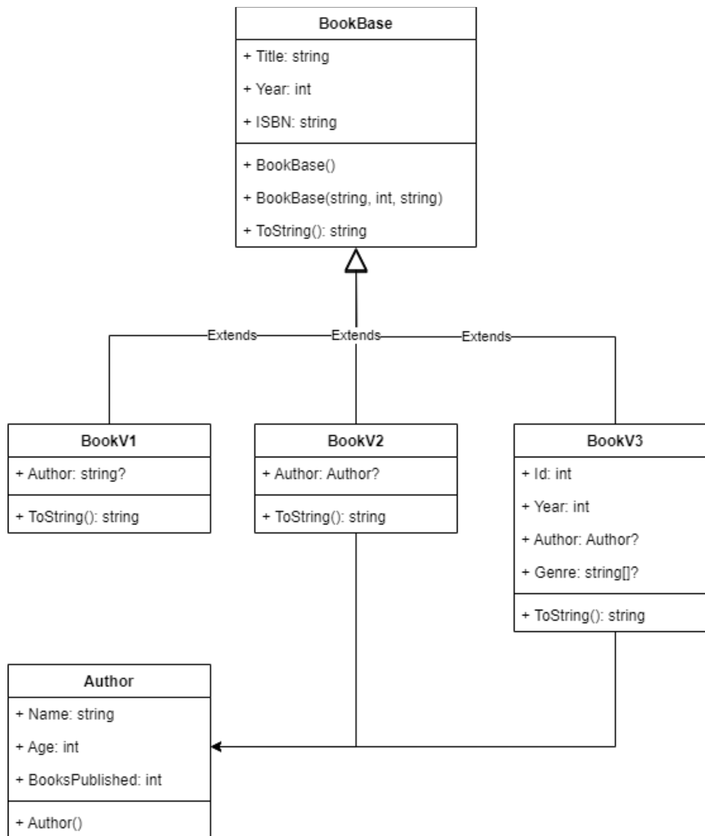
Kuva 17. Ohjelman aloituspiste.

Nyt olemme saaneet projektin luotua ja lähdemme tutustumaan datan muunnosongelmaan.

4.3.2 Datan mallinnus

Jotta voimme työskennellä kolmen eri kirjarajapinnan datan kanssa, joudumme mallintamaan jokaisen rajapinnan kirjadatan ohjelmaamme. Tämä luonnistuu tutkimalla rajapintojen tarjoaman datan ja luomalla dataa vastaavia tietomalliin kuuluvia luokkia.

Kuvan 18 luokkakaaviosta näemme, miten data on mallinnettu luokiksi. BookBase toimii yliluokkana, jonka kaikki kirjat perivät. BookBase sisältää kirjoille yhteisen datan ja sen avulla pystymme liikuttamaan ohjelman sisällä kirjojen dataa helposti. BookV1 esittää kirjakauppa sovellusrajapinnan v1 kirjadataa. Vastaavasti BookV2 ja BookV3 luokka esittävät kirjakaupparajapinnan v2 ja v3 kirjadataa.



Kuva 18. Kirjojen data luokkina luokkakaaviossa.

Luomme kuvan 20 mukaiset luokat projektiin ja mahdollistamme sarjallistamisen JSON-muotoon. Tämä onnistuu esimerkiksi käyttämällä Newtonsoft.Json .Net Nugetia, joka on hyvin suosittu, JSON-sarjoitin teknologia [22]. Tämän teknologian avulla Ilmoittamme kuvan 19 mukaisesti, mitkä kentät ovat JSON-ominaisuuksia.

```
13 references
public class BookBase
{
    [JsonProperty("title")]
    18 references
    public string Title { get; set; }
    [JsonProperty("year")]
    13 references
    public int Year { get; set; }
    [JsonProperty("isbn")]
    18 references
    public string ISBN { get; set; }

    0 references
    public BookBase()
    {
        Title = "";
        Year = -1;
        ISBN = "";
    }
    0 references
    public BookBase(string title, int year, string isbn)
    {
        Title = title;
        Year = year;
        ISBN = isbn;
    }

    3 references
    override public string ToString()
    {
        return $"Title: {Title}, Year: {Year}, ISBN: {ISBN}";
    }
}
```

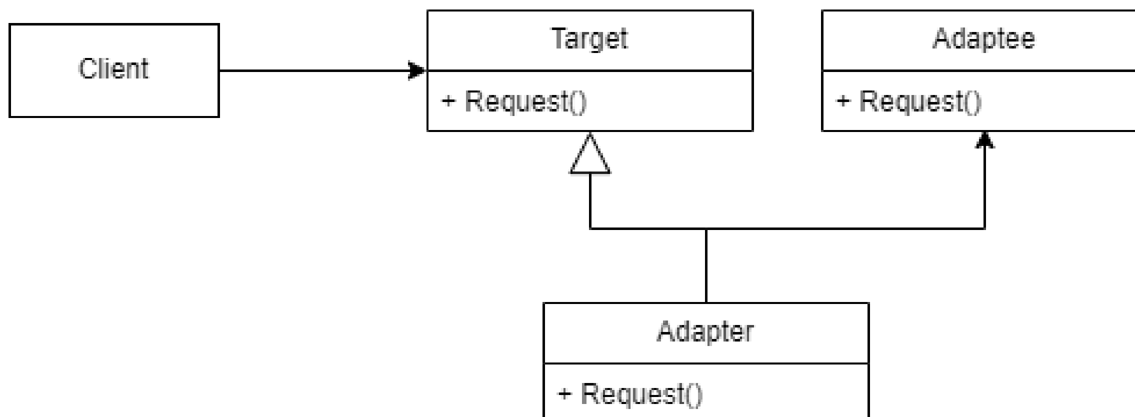
Kuva 19. BookBase-luokkakoodi.

Kuvassa 19 title-, year- ja isbn-kentät on merkattu JsonProperty-annotaatioiksi. Annotaatiot ovat metadataa Newtonsoft.Json .Net Nuget -kirjastolle, jotta se pystyy sarjallistamaan luokan JSON-dataksi ja JSON-datasta takaisin luokaksi.

4.3.3 Datan muuntaminen

Kirjojen datan mallinnuksen jälkeen voimme lähteä miettimään, miten pystymme muuntamaan datan toiseksi kirjadataksi. Tähän sopii erittäin hyvä suunnittelumalli nimeltä sovitin (adapter). Sovitin suunnittelumallilla pystymme luomaan jokaiselle kirjalle oman sovittimen, joka pystyy muuntamaan kirja datan toiseksi kirjadataksi. Kuvasta 20 näemme tyypillisen sovittimen toteutuksen.

Meillä on kohde (Target), johon haluamme muuntaa luokan, sovitin (Adapter) ja luokka, joka halutaan sovittaa (Adaptee).



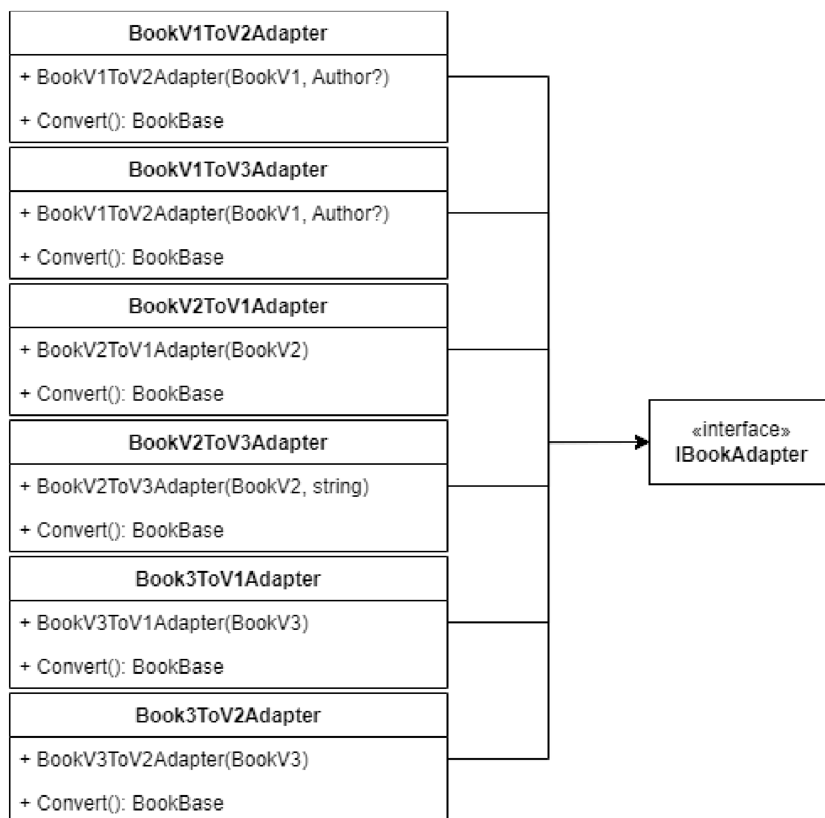
Kuva 19. Sovittimen tyypillinen toteutus.

Luomme kuvan 21 mukaisesti sovittimet jokaiselle -model luokalle. Näitä luokkia tulee yhteensä 6 kappaletta. Luokkien pitää toteuttaa IBookAdapteri-rajapintaa, jossa määritellään Convert-metodi. Metodissa luomme ja palautamme kohdekirjan, johon haluttu kirja on muunnettu.

```
3 references
public class BookV1ToV2Adapter : IBookAdapter
{
    private readonly BookV1 _bookV1;
    private readonly Author? _extraAuthorData;
    1 reference
    public BookV1ToV2Adapter(BookV1 bookV1, Author? extraAuthorData = null)
    {
        _bookV1 = bookV1;
        _extraAuthorData = extraAuthorData;
    }
    0 references
    public BookV1ToV2Adapter(BookV1 bookV1)
    {
        _bookV1 = bookV1;
    }

    6 references
    public BookBase Convert()
    {
        return new BookV2
        {
            Title = _bookV1.Title,
            Year = _bookV1.Year,
            ISBN = _bookV1.ISBN,
            Author = _extraAuthorData ?? new Author { Name = _bookV1.Author, Age = -1, BooksPublished = -1}
        };
    }
}
```

Kuva 21. Sovitinluokka BookV1-luokan mukaisen datan muuntamiseksi BookV2-luokan mukaiseksi.

Sovitinsuunnittelumalli toimii tässä pienessä kirjakauppa tapauksessa vielä erittäin hyvin, mutta luokkakaavio kuvasta 22 pystymme päättelemään, että isommassa skaalassa tulemme näkemään luokkien määrän kasvavan eksponentiaalisesti. Eksponentiaalinen kasvu tulee johtumaan siitä, että meillä on tarve muuntaa dataa molempiin suuntiin eli BookV1 halutaan muuntaa BookV2:ksi ja BookV2 halutaan muuntaa BookV1:ksi ja joudumme luomaan omat sovittimet jokaiselle datamuunnokselle.

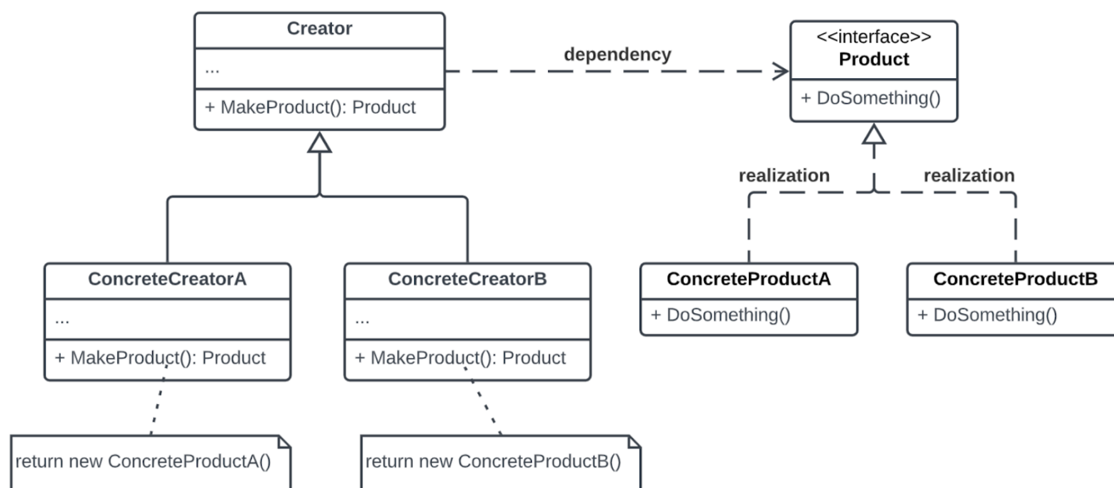


Kuva 20. Sovittimien luokkakaavio.

Tähän ongelmaan on useita mahdollisia ratkaisuja. Esimerkiksi voisimme luoda yhden yleisen dataluokan, johon muunnamme datan, jolloin meidän ei tarvitse muuntaa dataa molempiin suuntiin. Voisimme myös hylätä koko sovittimen ja siirtyä kartoituskirjastoihin, jotka automaattisesti pystyvät muuntamaan suurimman osan datasta automaattisesti ja loput pienellä hienosäätämisellä. Näitä ratkaisuita emme kuitenkaan tule käyttämään, koska projektin aloituksessa totesimme, että säilytämme datan saman näköisenä riippumatta, mistä kirjakaupan API-versiosta haemme datan ja automaattinen kartoitus piilottaa hyvin paljon siitä, mitä datan muuntaminen vaatii.

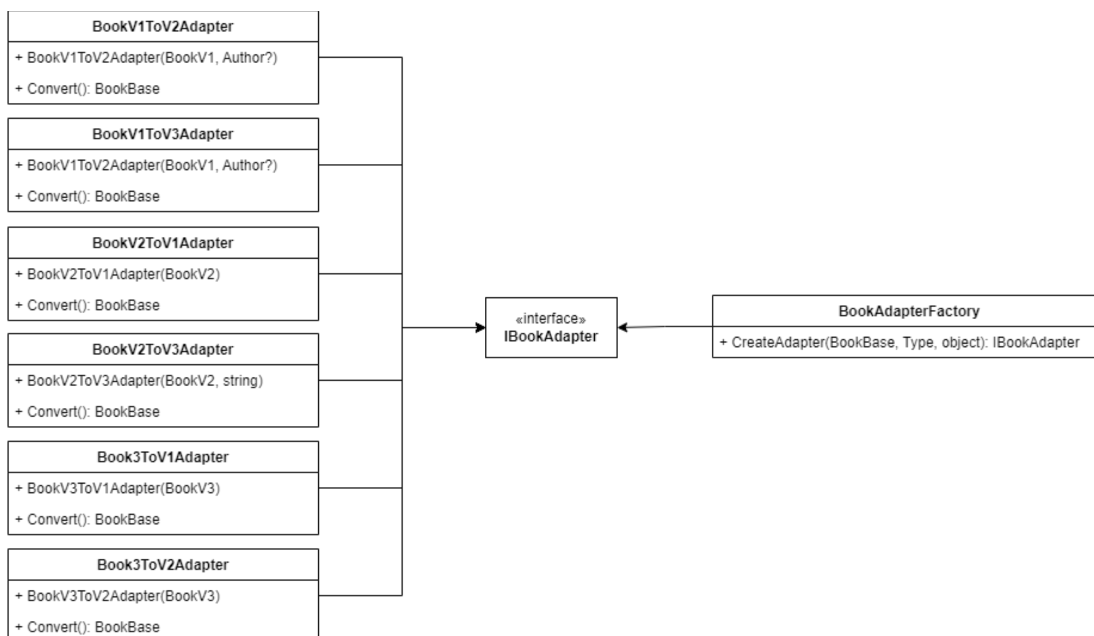
Seuraavaksi lähdemme vastaamaan kysymykseen siitä, miten voimme helposti valita oikean sovittimen. Tähänkin on olemassa todella hyvä suunnittelumalli nimeltään tehdasfunktio. Tehdasfunktiolta pystymme pyytämään olioita, ja se palauttaa meille halutun olion. Näin saamme yksinkertaistettua valintamekanismin ohjelmassamme. Kuvasta 23 näemme yleisesti, miten tehdasfunktio toimii, eli

meillä on luoja (Creator,) jota todelliset luojat (ConcreteCreator) A ja B perivät ja luojalla on riippuvuus tuote (Product) rajapintaan.



Kuva 21. Tehdasfunktion luokkakaavio perustoiminnasta.

Tällä tehdasfunktiolla pystymme pyytämään halutun tuotteen A tai B. Kuvassa 24 on meidän oma toteutuksemme yksinkertaistettuna eli BookAdapterFactory on riippuvainen IBookAdapterista, ja CreateAdapter-metodissa luomme halutun sovittimen riippuen annetuista parametreistä.



Kuva 22. Tehdasfunktion ja sovittimien yhteistyötä kuvaava luokkakaavio.

Kuvassa 25 näkyy tehdasfunktiomme CreateAdapter-metodin toteutus. Metodissa tutkimme, mitkä ovat lähde- ja kohdeluokkien tyypit ja niiden pohjalta valitsemme ja palautamme halutun sovittimen.

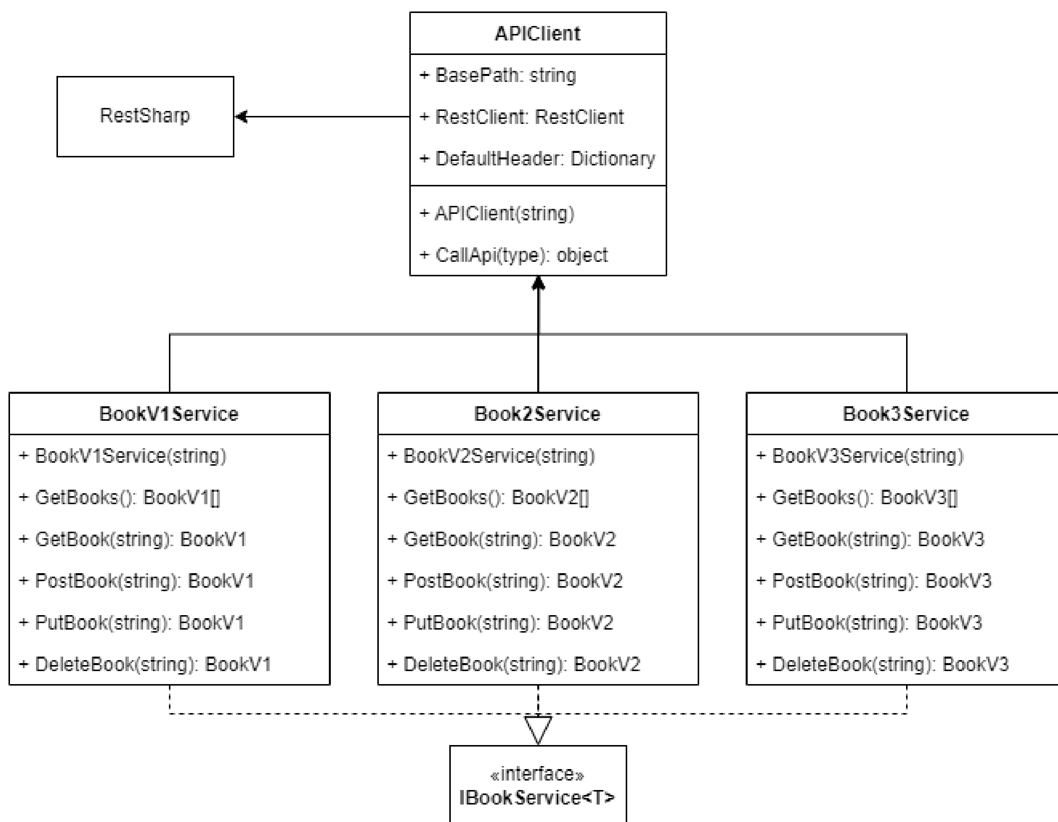
```
5 references
public static class BookAdapterFactory
{
    5 references
    public static IBookAdapter CreateAdapter(BookBase source, Type targetType, object extraData = null)
    {
        if (source == null)
        {
            throw new ArgumentNullException(nameof(source));
        }
        if (source is BookV1 && targetType == typeof(BookV2))
        {
            return new BookV1ToV2Adapter(source as BookV1, extraData as Author);
        }
        if (source is BookV1 && targetType == typeof(BookV3))
        {
            return new BookV1ToV3Adapter(source as BookV1, extraData as Author);
        }
        if (source is BookV2 && targetType == typeof(BookV1))
        {
            return new BookV2ToV1Adapter(source as BookV2);
        }
        if (source is BookV2 && targetType == typeof(BookV3))
        {
            return new BookV2ToV3Adapter(source as BookV2);
        }
        if (source is BookV3 && targetType == typeof(BookV1))
        {
            return new BookV3ToV1Adapter(source as BookV3);
        }
        if (source is BookV3 && targetType == typeof(BookV2))
        {
            return new BookV3ToV2Adapter(source as BookV3);
        }

        throw new NotImplementedException($"Adapter for converting {source.GetType().Name} to {targetType.Name} is not implemented.");
    }
}
```

Kuva 23. Tehdasfunktion toteutus.

4.3.4 Rajapintojen kutsuminen

Kun olemme luoneet kaikki model- ja adapteriluokat, voimme lähteä tekemään kirjakauppojen REST-rajapitojen palveluluokkia. Näitä luokkia tarvitsemme, jotta pystymme helposti tekemään kyselyjä rajapintoihin eri tavoilla.

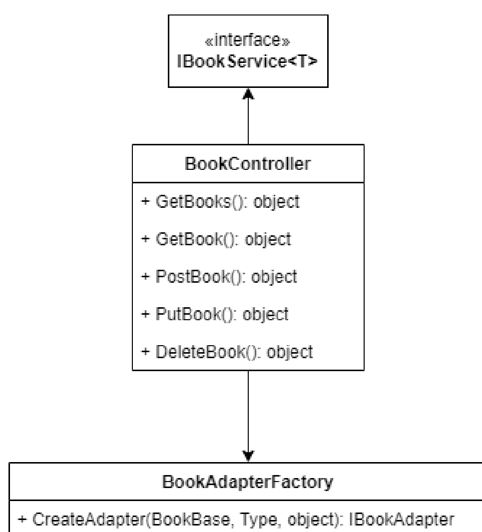
Luomme luokkakaavion kuvan 26 mukaisesti 4 luokkaa, joita ovat APIClient, BookV1Service, BookV2Service ja BookV3Service. APIClient on kääreluokka, jonka tehtävä on kääriä RestSharp .Net NuGetin toiminnallisuus yksinkertaisien metodien taakse. RestSharp on NuGetti, jolla pystyy tekemään helposti REST-rajapinta kutsuja.NET-sovelluksissa [23]. BookV1Service-, BookV2Service- ja BookV3Service-luokat käyttävät APIClientia ja toteuttavat IBookService-rajapintaa.



Kuva 24. Rajapintojen palveluluokkien luokkakaavio.

Jokainen Service-luokka palauttaa kirjojen tiedot siinä muodossa, missä rajapinnat antoivat datan.

Nyt kun meillä on mahdollisuus kutsua kaikkia kolmea kirjarajapintaa sekä vastaanottaa ja muunnella dataa, voimme yhdistää nämä kaikki elementit REST-kontrolleriluokassa. Luomme luokkakaavion 27 mukaisen BookController-luokan, joka perii AspNetCoren ControllerBase-luokkaa ja käyttää IBookService-rajapintaa ja BookAdapaterFactorya.



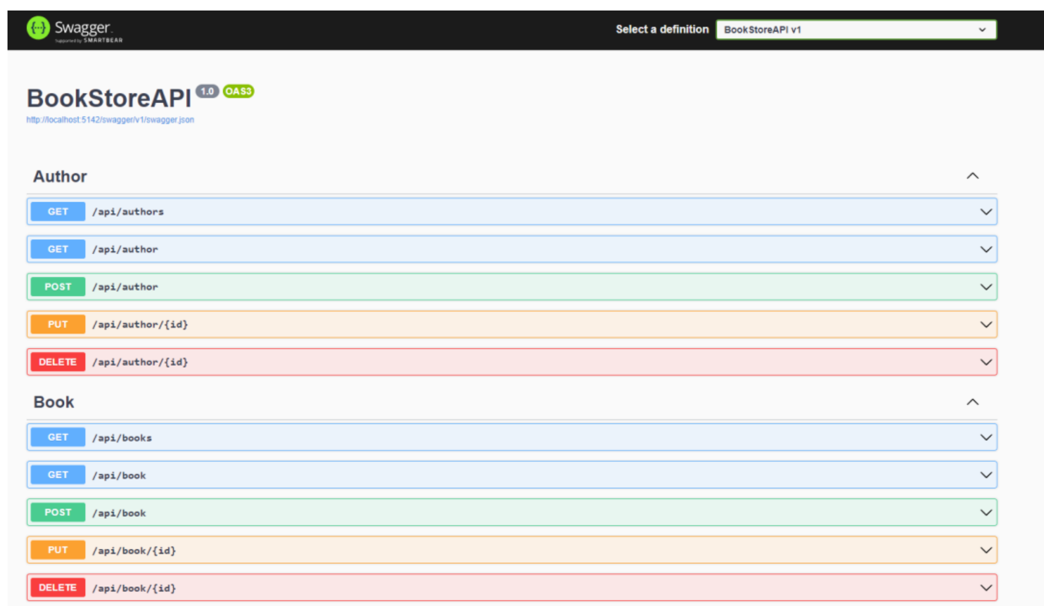
Kuva 25. BookControllerin luokkakaavio.

Tämän luomamme luokan avulla pystymme määrittämään REST-rajapintamme päätepisteet, joita voidaan kutsua. Jokainen päätepiste tarjoaa halutun kirjarajapinnan näköistä dataa. Kuvasta 28 pystymme näkemään esimerkkitoteutuksen GetBooks-metodista. Metodissa haetaan kaikista kirjarajapinnoista kaikki kirjat ja muunnetaan haluttuun datamuotoon.

```
[HttpGet("books", Name = "books")]
[Produces("application/json")]
0 references
public object GetBooks(string type)
{
    List<BookBase> books = new List<BookBase>();
    BookBase bookType;
    try
    {
        switch (type)
        {
            case "BookV1":
                bookType = new BookV1();
                break;
            case "BookV2":
                bookType = new BookV2();
                break;
            case "BookV3":
                bookType = new BookV3();
                break;
            default:
                return StatusCode(400);
        }
        foreach (var bookService in _booksServices)
        {
            bookService.GetBooks().ForEach(book =>
            {
                var adapter = BookAdapterFactory.CreateAdapter(book, bookType.GetType());

                var convertedBook = adapter.Convert();

                books.Add(book);
            });

        }
        return books;
    }
    catch (System.Exception)
    {
        return StatusCode(500);
    }
}
```

Kuva 26. GetBooks-metodin toteutus.

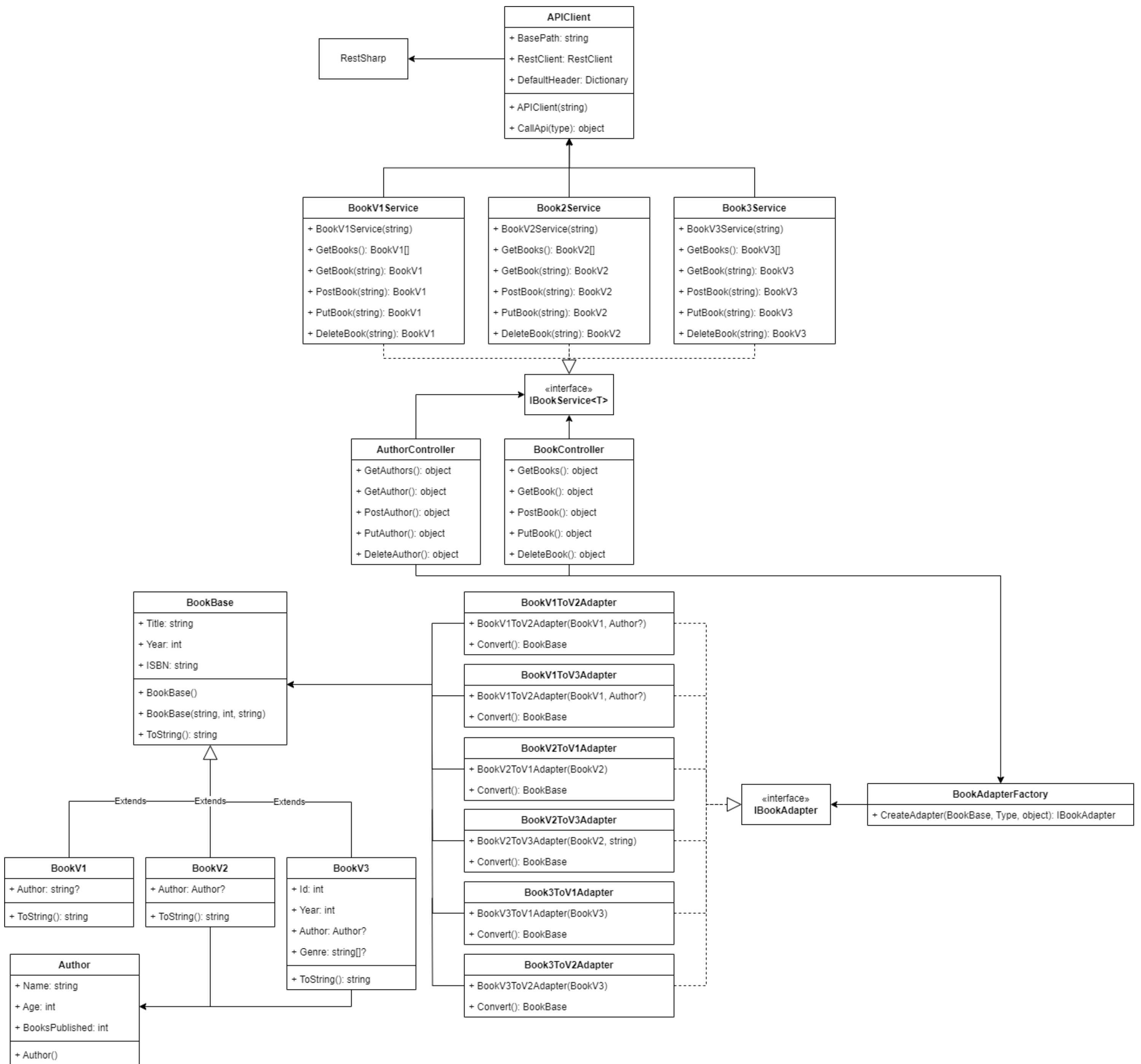
Kun olemme saaneet kaikki rajapinnan metodit luotua kontrollereihin, swagger automaattisesti pystyy suoraan koodista generoimaan meille visualisoinnin rakentamastamme REST-rajapinnasta. Kuvassa 29 näkyvät kaikki rajapinnan päätepisteet swagger UI:ssa.



Kuva 29. Rajapinnan visualinen esitys Swagger UI:ssa.

Voimme näitä päätepisteitä testailla ja hyödyntää tulevissa firman sovelluksissa. Tämän REST-rajapinnan voimme nyt julkaista halutulle palvelimelle käyttäen dokkerin säiliötä. Tämän kontin luominen hyvin automaattinen, koska valitsimme dokkerin projektin aloituksessa.

Kuvassa 30 näemme koko ohjelman luokkakaavion, joka muodostui kehityksen aikana. Tämä luokkakaavio hyvin havainnollistaa järjestelmän rakenteen ja eri komponenttien väliset yhteydet, mikä näyttää paremman kokonaiskuvan.



Kuva 30. Demo ohjelman luokkakaavio.

5 Yhteenveto

Tämä insinöörityö käsitteli välityspalvelimen suunnittelua ja toteutusta API-versioiden hallintaan ja yhteensopivuuksien varmistamiseen. Työn lähtökohtana on kuvitteellisen yrityksen ongelma, jossa vanhat sovellukset eivät toimi uusien eri API-versioiden kanssa, mikä aiheuttaa merkittäviä päivitystarpeita ja resurssien käyttöä. Ratkaisuna esitellään välityspalvelin, joka toimii sovellusten ja API-palveluiden välissä, mikä mahdollistaa vanhojen versioiden käytön muuntamalla ja emuloimalla poistuneiden API-versioiden vastauksia.

Kävimme läpi välityspalvelimien yleistä toimintaa sekä mitä ovat välittävät ja käänteiset välityspalvelimet. Lisäksi työssä tarkasteltiin API-yhdyskäytävää, joka yhdistää useita mikropalveluita ja helpottaa sovellusten hallintaa ja kehitystä. Demon kehityksessä käytetiin teknologioita, kuten Docker, REST-rajapinnat, Swagger ja .NET Core.

Lähteet

1 Martin Sysel, Ondřej Doležal. 2014. An Educational HTTP Proxy Server. Verkkoaineisto. Researchgate <https://www.researchgate.net/publication/261104026_An_Educational_HTTP_Proxy_Server> Luettu 15.7.2024.

2 Robin Geuens. 2024. The Different Types of Proxies and Their Uses. Verkkoaineisto. Soax. <https://soax.com/blog/types-of-proxies> Päivitetty 15.2.2024. Luettu 15.7.2024.

3 Lisa Whelan. 2024. What is a forward proxy and how do they work? Verkkoaineisto. Soax. <https://soax.com/blog/forward-proxies> Päivitetty 23.1.2024. Luettu 15.7.2024.

4 Michael Buckbee. 2024. What is a Proxy Server and How Does it Work? Verkkoaineisto. Varonis. <https://www.varonis.com/blog/what-is-a-proxy-server> Päivitetty 24.6.2024. Luettu 20.7.2024.

5 Maria Kazarez. 2022. What is a Reverse Proxy and How Does It Work? Verkkoaineisto. Soax. < https://soax.com/blog/what-is-a-reverse-proxy-and-how-does-it-work> Päivitetty 14.12.2022. Luettu 20.7.2024.

6 James Montgomery, Paul Kirvan. 2022. API gateway (application programming interface gateway). Verkkoaineisto. Techtarget. <https://www.techtarget.com/whatis/definition/API-gateway-application-programming-interface-gateway> Päivitetty 6.2.2024. Luettu 3.8.2024.

7 Pulak Das. 2023. Introduction to API Gateway in Microservices Architecture. Verkkoaineisto. Imesh <https://imesh.ai/blog/introduction-to-api-gateway-in-microservices-architecture/> Päivitetty 20.8.2024. Luettu 4.8.2024.

8 Adam Gluck. 2020. Introducing Domain-Oriented Microservice Architecture. Verkkoaineisto. Uber <https://www.uber.com/en-FI/blog/microservice-architecture/ > Luettu 10.8.2024.

9 Mehmet Ozkaya. 2021. API Gateway Pattern. Verkkoaineisto. Medium <https://medium.com/design-microservices-architecture-with-patterns/api-gateway-pattern-8ed0ddfce9df> Luettu 10.8.2024.

10 Erich Gamma, Richard Helm, ym. 1994. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. S21-23.

11 Sonu Kumar. 2024. What is Docker and why do we use it? Verkkoaineisto. Medium. <https://medium.com/@me.sonu300/what-is-docker-and-why-do-we-use-it-c86b11559b3b> Luettu 17.8.2024.

12 Stephanie Susnjara, Ian Smalley. 2024. What is Docker? Verkkoaineisto.
IBM. <https://www.ibm.com/topics/docker> Luettu 17.8.2024.

13 Di Marco. 2022. Docker: terminologia base. Verkkoaineisto. Betaingegne-
ria. <https://betaingegneria.it/2022/08/12/docker-terminologia-base/> Lu-
ettu 17.8.2024.

14 Roy Thomas Fielding. 2000. Architectural Styles and the Design of Net-
work-based Software Architectures. Verkkoaineisto. UNIVERSITY OF CA-
LIFORNIA, IRVINE. <https://ics.uci.edu/~fielding/pubs/disserta-
tion/rest_arch_style.htm> Luettu 31.8.2024.

15 Vijay Surwase. 2016. REST API Modeling Languages - A Developer's
Perspective. Verkkoaineisto. Betaingegneria. <https://www.ijste.org/arti-
cles/IJSTEV2I10199.pdf > Luettu 31.8.2024.

16 Jéssica Soares Dos Santos, Leonardo Guerreiro Azevedo, Elton F. S.
Soares, ym. 2020. Analysis of Tools for REST Contract Specification in
Swagger/OpenAPI. Verkkoaineisto. IBM Research. <https://www.sci-
tepress.org/Papers/2020/93812/93812.pdf> Luettu 7.9.2024.

17 SmartBear Software. 2024. OpenAPI Specification. Verkkoaineisto. Swag-
ger. <https://swagger.io/specification/> Luettu 8.9.2024.

18 Andrea Chiarelli. 2021. What is .NET? An Overview of the Platform. Verk-
koaineisto. Auth0. <https://auth0.com/blog/what-is-dotnet-platform-over-
view/> Päivitetty 15.10.2021. Luettu 8.9.2024.

19 Visual Studio 2022. Verkkoaineisto. <https://visualstudio.micro-
soft.com/vs/>. Luettu 6.10.2024.

20 Microsoft Learn. Verkkoaineisto. <https://learn.microsoft.com/en-us/visu-
alstudio/extensibility/internals/solution-dot-sln-file?view=vs-2022>Luettu
7.10.2024.

21 Microsoft Learn. Verkkoaineisto. <https://learn.microsoft.com/en-us/dot-
net/csharp/tutorials/top-level-statements>. Luettu 8.10.2024.

22 Json.NET - Newtonsoft. Verkkoaineisto. <https://www.newton-
soft.com/json>. Luettu 9.10.2024.

23 RestSharp. Verkkoaineisto. <https://restsharp.dev>. Luettu 10.10.2024.