Miko Parkkinen

# WebAPI:n toteutus asiakastulosteiden hallintaan

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

4.5.2018

Metropolia

| | |
|---|---|
| Tekijä<br>Otsikko<br><br>Sivumäärä<br>Aika | Miko Parkkinen<br>WebAPI:n toteutus asiakastulosteiden hallintaan<br><br>35 sivua<br>4.5.2018 |
| Tutkinto | insinööri (AMK) |
| Tutkinto-ohjelma | tieto- ja viestintätekniikka |
| Ammatillinen pääaine | pelisovellukset |
| Ohjaajat | lehtori Ilpo Kuivanen<br>Kojamo Oyj tietohallinnon palvelupäällikkö Piia Hyvärinen |

Insinöörityönä toteutettiin palvelu yritykselle Kojamo Oyj. Toteutettu palvelu sai nimekseen PdfLetterService, ja se vastaa yrityksestä lähtevien PDF-muotoisten E-kirjeiden lähetyksestä. Lähteviä PDF-muotoisia E-kirjeitä ovat esimerkiksi sopimus- ja hakemusvahvistukset sekä maksumuistutukset.

Palvelu korvaa aiemman kirjeiden lähetykseen käytetyn sovelluksen. Tavoitteena oli toteuttaa toimintavarma ja skaalautuva kokonaisuus, joka suoriutuu raskaistakin lähetyskuormista. Lisäksi palvelun haluttiin pystyvän lähettämään kirjeitä ajastetusti haluttuun aikaan.

PdfLetterService toteutettiin käyttämällä ASP.NET Web Api -viitekehystä. PdfLetterService hyödyntää toiminnassaan pilvipalveluita, ja se on itsekin hostattuna pilveen. Käytetty pilvipalveluntarjoaja on Microsoft Azure. Microsoft Azuren tarjoamista palveluista PdfLetterService käyttää mm. seuraavia: Azure App Service, Azure Cosmos DB, Azure Blob Storage ja Azure Table Storage.

Työlle arvioitu toteutusaika saavutettiin. PdfLetterServicen toteutus kesti yhteensä n. 5 viikkoa, joista jokainen viikko sisälsi 37,5 h työtä. Raportin kirjoittamista ei laskettu mukaan kuluneeseen aikaan.

Lopputuloksena saatiin valmiiksi toimiva palvelu. PdfLetterService on päivittäisessä käytössä yrityksen sovellusten tuotantopuolella. Aiempaan kirjeiden lähetyssovellukseen verrattuna PdfLetterService säästää aikaa ja rahaa sekä selkeyttää lähetettävien kirjeiden esikatselua.

| | |
|---|---|
| Avainsanat | ASP.NET, Web API, Microsoft Azure |

| | |
|---|---|
| Author<br>Title<br><br>Number of Pages<br>Date | Miko Parkkinen<br>WebAPI:n toteutus asiakastulosteiden hallintaan<br><br>35 pages<br>4.5.2018 |
| Degree | Bachelor of Engineering |
| Degree Programme | Information and Communication Technology |
| Professional Major | Game Applications |
| Instructors | Ilpo Kuivanen, Senior Lecturer<br>Piia Hyvärinen, Kojamo Oyj IT management Service Manager |

The purpose of this Bachelor's thesis was to produce a service, which handles the sending process of PDF-form electrical letters. For example, the service will send PDF-form contract verifications and reminders.

The project was produced to a company called Kojamo Oyj. The service was named as PdfLetterService and it replaces an old system that the company has been using to send electrical letters. The goals of this project were to create a service that is reliable and can be scaled to handle massive amounts of letter send requests. In addition, PdfLetterService was also required to be able to send scheduled letters.

PdfLetterService was developed using ASP.NET Web API framework. The advantage of cloud services were a major part of PdfLetterService's development. Microsoft Azure was used as the provider of these cloud services. PdfLetterService used among other things these Microsoft Azure services: Azure App Service, Azure Cosmos DB, Azure Blob Storage and Azure Table Storage.

Estimated schedule of the project was 5 weeks and each week including 37.5 working hours. The project was completed on time. Writing process of the raport of this thesis was not included in this timeline.

In conclusion, the project was a success. PdfLetterService is being used on daily basis to send electrical letters. Compared to the previous system PdfLetterService saves the companys time, money and trouble.

| | |
|---|---|
| Keywords | ASP.NET, Web API, Microsoft Azure |

## Sisällys

## Lyhenteet ja termit

| | |
|---|---|
| API | Application Programming Interface. Ohjelmointirajapinta, jota eri sovellukset voivat käyttää tiedonvaihtoon. |
| NoSQL | Not only SQL. Käsite, jolla kuvataan perinteisestä relaatiomallista eroavaa tietokantaa. NoSQL-tietokanta ei noudata perinteisten tietokantojen tapaista taulukkoskeemaa. |
| JSON | JavaScript Object Notation. Avoimen standardin tiedostomuoto tiedonvälitykseen. Nimestään huolimatta se on JavaScriptistä riippumaton. |
| Azure | Microsoftin kehittämä pilvipalvelualusta. Azure tarjoaa erilaisia PaaS (Platform as a Service) sekä IaaS (Infrastructure as a Service) palveluita. |

# 1 Johdanto

Tämä raportti kertoo Kojamo Oyj:lle tehdyn Web API:n (Application Programming Interface) toteutuksesta. Kojamo Oyj keskittyy asuntosijoittamiseen Suomessa ja on 4,3 miljardin euron asunto-omaisuuden arvollaan mitattuna Suomen suurin kiinteistösijoittaja [1]. Kojamo Oyj:n tietohallinnon sovelluskehitystiimiin kuuluu seitsemän henkilöä. Sovelluskehitystiimi on kehittänyt yrityksen tärkeimmät sovellukset vuosien aikana. Vuoteen 2017 asti Kojamo Oyj on tunnettu nimellä VVO-yhtymä Oyj.

Toteutettu tuote/projekti kantaa nimeä ”PdfLetterService”. PdfLetterService käsittelee yrityksestä lähteviä PDF-muotoisia E-kirjeitä. Tällaisia kirjeitä voivat olla esimerkiksi erilaiset vahvistukset ja maksumuistutukset. PdfLetterService integroitiin osaksi yrityksen tiedonvälitysjärjestelmää.

Pilvipalveluratkaisut ovat tärkeä osa PdfLetterServicen toimintaa, joten tässä raportissa on kerrottu kattavasti projektissa käytetyistä Microsoft Azuren pilvipalveluista. Pilvipalveluiden hyödyntäminen takaa tuotteen skaalautuvuuden sekä toimivuuden myös tulevaisuudessa.

PdfLetterServicen toteutus on kauttaaltaan asynkroninen. Raportissa perehdytään hieman syvemmin asynkronisen sekä synkronisen toteutuksen hyödyn eroihin. Vertailtavana ovat PdfLetterServicen toteutus asynkronisena sekä synkronisena versiona. Asynkronisen toteutuksen myötä pilvipalveluiden skaalautuvuudesta saadaan maksimaalinen hyöty irti.

Ohjelmassa mahdollisesti tapahtuvien virheiden käsittely ja kirjaaminen ovat tärkeitä asioita hyvin toimivan ohjelman kannalta. Raportti kertoo, miten PdfLetterService hoitaa virheiden käsittelyn sekä tiedon kirjaamisen. Tarkoituksena on, että tieto tapahtuneista virheistä sekä epäonnistuneista kirjeiden lähetyksistä on helposti ja nopeasti saatavilla.

Raportti kertoo myös, miten PdfLetterServicen testaus tehtiin. Testaus on olennainen osa ohjelmistokehitystä, joten PdfLetterServicen testaukseen käytettiin erilaisia työkaluja sekä itse toteutettua TestClient-ohjelmaa.

# 2 Palvelun suunnittelu

## 2.1 Vaatimukset

PdfLetterServicen tulee vastaanottaa E-kirjeen data ja tallentaa siitä kaikki tiedot tietokantaan. Mahdollisesti kirjeen mukana tuleva PDF-tiedosto tallennetaan eri paikkaan. Tietokantaan tallennettu kirje sisältää url-linkin, jolla PDF-tiedoston voi hakea erillisestä tallennuspaikasta.

PdfLetterServicen vastaanottama data on JSON-muotoista. Vastaanotettava JSON sisältää myös mahdollisen PDF-tiedoston, joka on koodattuna Base64-muotoon. Tallennetun datan ja PDF:n haku tietokannoista pitää olla nopeaa ja tehokasta. Data pitää pystyä poistamaan niin, että kaikki tiedot, mukaan lukien PDF-tiedosto poistuvat. Poistamisen lisäksi kirjeen lähetys pitää pystyä perumaan, mikäli kirjettä ei ole vielä lähetetty eteenpäin. Perumisen jälkeen kaikki kirjeen data poistuu.

Kirjeen lähetysaika on oltava määriteltävissä. Lähetysaika vastaanotetaan muun saapuvan datan mukana. Mikäli lähetysaikaa ei ole annettu, se asetetaan automaattisesti eteenpäin määritetyn ajan verran.

PdfLetterServicen pitää pystyä käsittelemään ja lähettämään eteenpäin vähintään 6000 kirjettä yhden yön aikana. Kaikkien kirjeiden on lähdettävä eteenpäin. Mikäli kirjeen lähetys jostain syystä epäonnistuu, siitä on oltava tieto saatavilla.

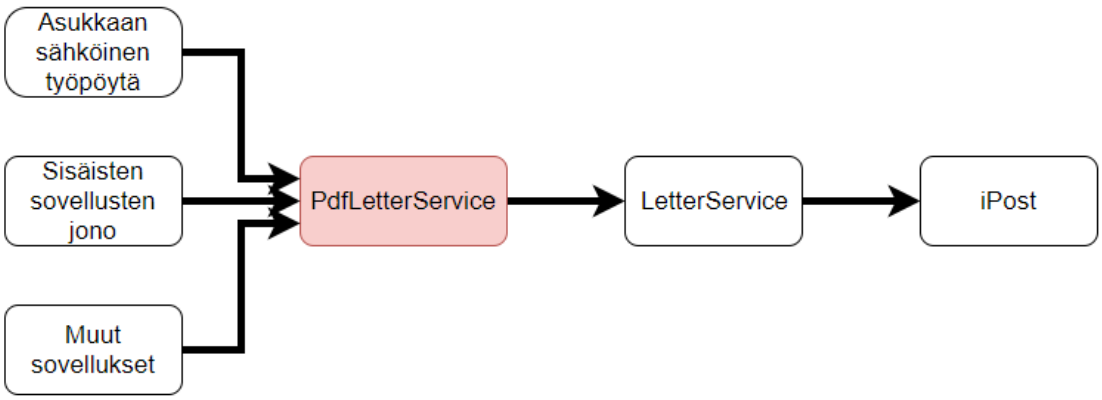
## 2.2 Käyttötarkoitus

PdfLetterService korvaa yrityksen vanhan E-kirjeiden lähettämiseen tarkoitetun ratkaisun. Vanha ratkaisu lähetti kirjeen eteenpäin postille tekstimuodossa. Postin päässä oli malli, jossa tekstit laitettiin oikeille paikoilleen. Näin muodostui A4-kokoinen kirje. Mikäli postin päässä olevaa mallia haluttiin muuttaa, esimerkiksi teksti alkamaan eri kohdasta,

se maksoi satoja euroja. Muutoksen jälkeen vanhaa mallia ei enää voinut käyttää ja muutoksesta aiheutui turhaa vaivaa.

PdfLetterService selkeyttää kirjeen lähettäjän toimintaa. Kirjeen tekstin asettelua ja kokoa pystyy muuttamaan ilman lisäkustannuksia. PDF-muotoisesta E-kirjeestä on helppo katsoa, miten teksti on asettautunut kirjeeseen ja nähdä mahdolliset asetteluvirheet. Vanhassa ratkaisussa kirjettä ei pystynyt esikatselemaan, koska kirje koottiin vasta postin päässä sinne toimitetun tiedon perusteella.

PdfLetterService integroidaan osaksi yrityksen tiedonvälitysinfrastruktuuria. PdfLetterServicen kautta kulkevat kaikki yrityksestä lähtevät E-kirjeet. Lähteneet kirjeet tallennetaan niin, että ne ovat myöhemmin luettavissa.




Kuva 1. PdfLetterService kirjeen lähetyksen aikana.

Vanha ratkaisu oli koodattu suoraan yhteen ohjelmaan. Koska PdfLetterService toimii palveluna pilvessä, se ei ole sidottu vain yhden ohjelman käytettäväksi. Näin ollen tulevaisuudessa lukemattomat järjestelmät pystyvät hyödyntämään PdfLetterServicen kirjeenlähetystoimintoja.

# 3 Palvelun toteutus

## 3.1 Käytetyt menetelmät ja palvelut

### 3.1.1 ASP.NET Web Api

ASP.NET Web Api on viitekehys, joka mahdollistaa web API:n (application programming interface) toteuttamiseen .NET-viitekehyksen päälle. Yleisin syy käyttää ASP.NET Web Api -viitekehystä on silloin, kun halutaan tehdä RESTful (Representational State Transfer) API. Käytännössä REST on toteutusmalli, joka asettaa tiettyjä rajoitteita toteutukselle. Rajoitteita ovat esimerkiksi seuraavat neljä asiaa [2]:

- Toteutus noudattaa Client-server-arkkitehtuuria, jolla halutaan jakaa asiakkaan (client) ja palvelun (server) toiminnot selvästi erilleen.
- Toteutus on tilaton (stateless). Tällä tarkoitetaan sitä että, palvelu ei tallenna mitään tietoja asiakkaasta. Asiakkaalta saapuvat pyynnöt sisältävät aina kaiken tarvittavan, jota palvelu tarvitsee suorittaakseen pyynnön loppuun.
- Toteutus hyödyntää vastausten välimuistitusta (cacheability). Vastaus, jonka tietosisältö ei tule muuttumaan, voidaan merkitä ”cacheableksi”, jolloin vastaanottaja tietää, että tätä asiaa ei tarvitse kysyä heti uudelleen.
- Toteutuksen rajapinta on yhdenmukainen (uniform). Resurssien hakemiseen ja muokkaamiseen käytetään HTTP-metodeja GET, POST, PUT jne.

PdfLetterService on RESTful web API, joka on toteutettu edellä mainitulla ASP.NET Web API viitekehyksellä. Käytetty ohjelmointikieli on C#.

Yksi .NET-viitekehyksen tarjoamista hyödyistä on NuGet Package Manager. NuGet Package Managerin avulla voi asentaa projektiin paketteja, jotka tuovat ominaisuuksia, joita pelkkä .NET-viitekehys ei suoraan tarjoa. Kaikki PdfLetterServicen käyttämät ulkopuoliset paketit on asennettu käyttämällä NuGet Package Manageria. Tärkeimpiä käytettyjä paketteja ovat esimerkiksi:

- Microsoft.Azure.DocumentDB, jota käytetään tietokantaoperaatioiden suorittamiseen.
- Newtonsoft.Json, jota käytetään luokkien ja datan serialisointiin JSON:sta tai JSON:ksi.
- Swashbuckle, jota käytetään Swagger-apityökalun ajamiseen.

- log4net ja log4net.Appender.Azure, joita käytetään lokien kirjaamiseen ja tallennukseen.

Pakettien asentaminen projektiin NuGet Package Managerin avulla on nopeaa ja vaivatonta. Valittavissa ovat myös kaikki julkaistut versiot asennettavasta paketista sekä mahdolliset beta-versiot.

### 3.1.2 Azure API Management

Azure API Management on monen eri API:n hallinnoimiseen tarkoitettu palvelu [3]. Se tarjoaa yhdyskäytävän (API Gateway), jolloin monta eri API:a voidaan laittaa yhden yhdyskäytävän taakse. Silloin yhteydenpito API:hin tapahtuu saman yhdyskäytävän kautta ja url-osoite on kaikilla API:lla muotoa https://accountname.azure-api.net/apiname. API Management voidaan asettaa vaatimaan kaikilta saapuvilta pyynnöiltä avain ”API Management Subscription Key”, jota ilman pyyntö hylätään välittömästi. Tämä on kätevä tapa autentikoida saapuvia pyyntöjä. Myös liikenteen rajoittaminen tiettyä avainta kohden on hyödyllinen ominaisuus tarvittaessa rajoittamaan liikaa liikennettä. Avaimia pystyy hallinnoimaan ja generoimaan API Managementin Portalista.

API Managementiin on mahdollista asettaa käytäntöjä (policy), joilla tutkitaan ja muokataan saapuvaa dataa, ennen kuin se päätyy itse API:lle. PdfLetterService konfiguroitiin hyväksymään vain liikenne, joka tulee yhdyskäytävältä. Tässä onnistuttiin käyttämällä ”Mutual Certificate Authentication”-menetelmää. Mutual Certificate Authentication toimii käytännössä niin, että ensin API Managementtiin ladataan sertifikaatti. Sitten konfiguroidaan API Management lisäämään tämä sertifikaatti kaikkiin pyyntöihin, jotka se ohjaa PdfLetterServicelle. PdfLetterServicellä on myös tiedossa kyseinen sertifikaatti, joten saapuvasta pyynnöstä on helppo tarkastaa, vastaako pyynnön mukana tuleva sertifikaatti tiedossa olevaa sertifikaattia.

API Managementissa on kaksi puolta, ”Publisher portal” ja ”Developer portal”. Developer portalin puolelta kaikki käyttäjät, joilla on tiedossa API Management Subscription Key, voivat testata ja tehdä kutsuja API:lle.

PdfLetterServiceä hallinnoidaan Publisher portalin puolelta. Publisher portaliin laitetaan kuvaus kaikista operaatioista, joita API:a voi pyytää suorittamaan. Kuvauksiin liitetään

myös tarkat mallit siitä, millaisessa muodossa dataa luetaan sisään ja palautetaan takaisin kutsujalle. Näin ollen tarvittaessa Developer portalin puolelta muut kehittäjät, jotka eivät tiedä PdfLetterServicen toteutusta, voivat helposti katsoa tarvittavat datamallit integraatioita tai muuta tarkoitusta varten.

### 3.1.3 Azure App Service

PdfLetterService ajetaan Azuressa App Servicenä. App Service on Azuren tarjoama palvelu, joka mahdollistaa sovelluksen ajamisen pilvessä. Azure takaa App Service sovelluksille 99,95 %:n käytettävyysajan [4]. Käytettävyysajalla tarkoitetaan aikaa, jolloin palvelu on käytettävissä, eli ns. ”online”.

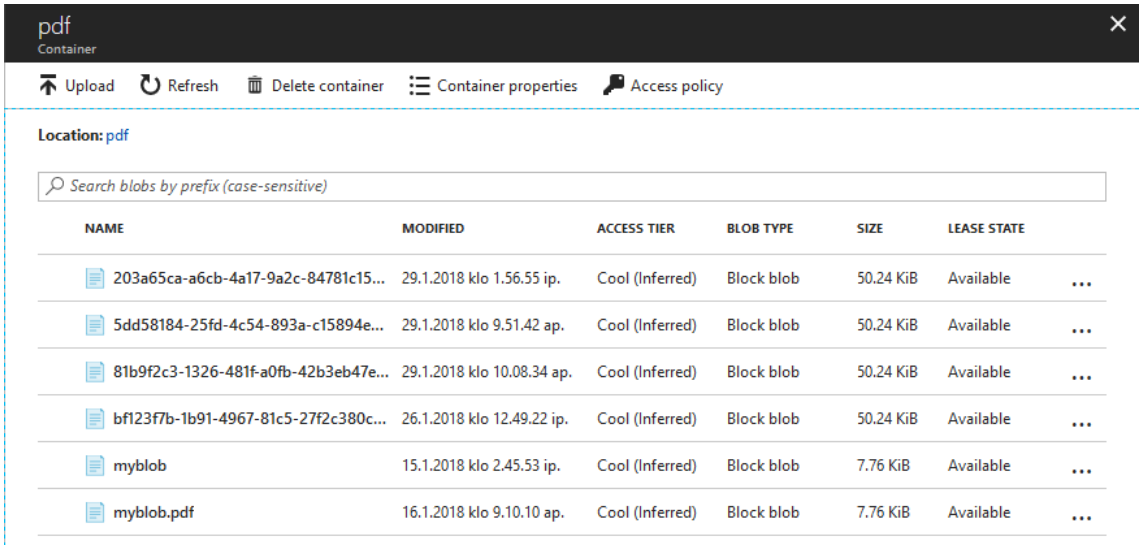
### 3.1.4 Azure Cosmos DB

Kirjeiden datan tallennuspaikaksi valittiin Azure Cosmos DB. Azure Cosmos DB mahdollistaa NoSQL-tietokantojen luomisen. PdfLetterServicessä käytettiin Cosmos DB SQL -rajapintaa, joka on tarkoitettu dokumenttien säilytykseen. Kaikki muu data paitsi PDF-tiedostot tallennettiin tietokantaan JSON-muotoisena. Käytetty Cosmos DB SQL -rajapinta mahdollistaa perinteisten Sql-komentojen ajamisen tietokantaa vasten. Azure takaa Cosmos DB:lle 99,999 %:n käytettävyysajan lukuoperaatioille sekä 99,99 %:n käytettävyysajan muille operaatioille [5]. Jos todellinen käytettävyysaika jää alle luvatun rajan, saa kuukausimaksusta hyvitystä vähintään 10 %.

### 3.1.5 Azure Blob Storage

Azure Blob Storage mahdollistaa suurien datamäärien (jopa eksatavujen [6]) säilytyksen tehokkaasti ja niin, että data on nopeasti saatavilla. Data voidaan tallentaa hot-, cold- tai archive-kerrokseen. Hot-kerrokseen on syytä tallentaa data, johon tarvitsee päästä käsiksi usein, ja cold-kerrokseen data, jota käytetään harvemmin. Hot-kerroksessa olevan datan säilyttäminen maksaa enemmän kuin cold-kerroksessa olevan datan. Siksi on hyvä miettiä, miten sijoittaa data eri kerrosten välillä. Blob Storageen tallennettua tiedostoa kutsutaan blobiksi. Kaikki blobit tallennetaan säilöihin (container). Yhdellä Blob Storage accountilla voi olla monia säilöjä, joihin on hyvä lajitella erilaiset datat kuten esimerkiksi PDF-tiedostot. Blobeille annetaan nimeksi jokin uniikki merkkijono, koska saman

säilön sisällä ei voi olla kahta samannimistä blobia. PdfLetterService antaa uusille blobeille nimeksi GUID (Globally Unique Identifier) -tyyppisen merkkijonon. Blobeille on myös mahdollista asettaa metadataa. Esimerkiksi kirje-id on yksi metadata tieto, joka tallennetaan kaikille blobeille. Näin ollen blobeista on helppo tunnistaa, mille kirjeelle ne kuuluvat.

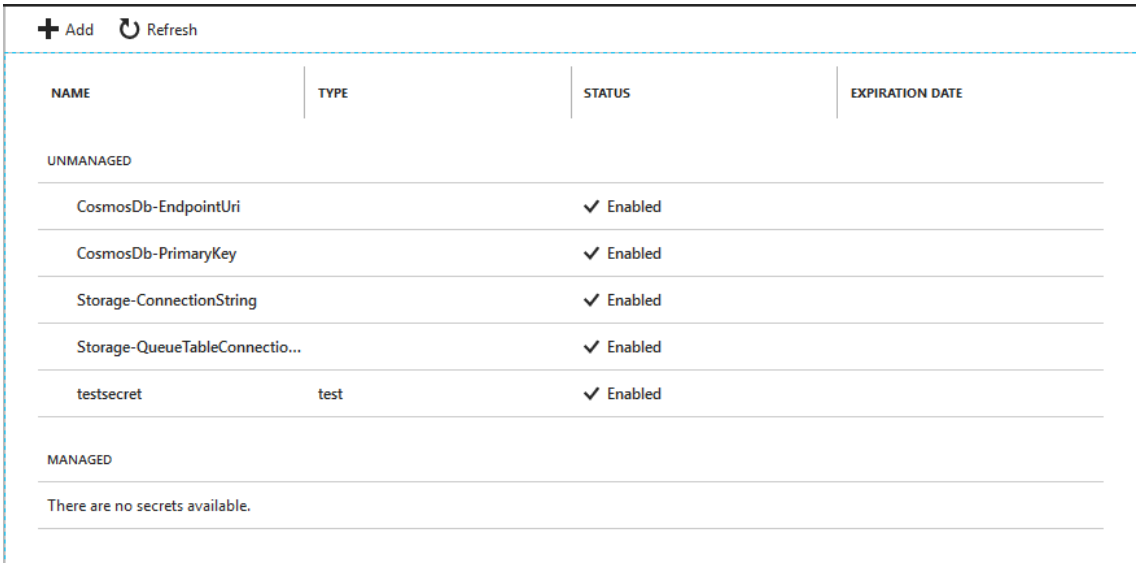



Kuva 2. Blob Storagen näkymä Azure Portalissa. Kuvassa pdf-nimisen säilön sisältö, jossa on kuusi tiedostoa.

### 3.1.6 Azure Key Vault

Azure Key Vault on nimensä mukaisesti palvelu, joka tarjoaa mahdollisuuden tallentaa avaimia (key) tai salaisuuksia (secret). Avaimet voivat olla esimerkiksi sertifikaattien pfx-avaimia. Salaisuudeksi voi tallentaa minkä tahansa vapaamuotoisen merkkijonon. Key Vaultissa olevat tiedot salataan niin, että edes Microsoft ei pääse näkemään niitä [7].

PdfLetterService käyttää Key Vaultia mm. connection stringien lukemiseen. Tämä tarkoittaa sitä, että arkaluontoisia avaimia ei tarvitse kova koodata suoraan sovellukseen, vaan ne pysyvät turvassa Key Vaultissa. Mikäli esimerkiksi Blob Storagen connection string joudutaan uusimaan, on helppoa, että se tarvitsee päivittää vain Key Vaultiin, josta PdfLetterService sen käy lukemassa. Ei tarvitse muuttaa itse sovelluksen koodia, ja säästytään turhilta käännösoperaatioilta.



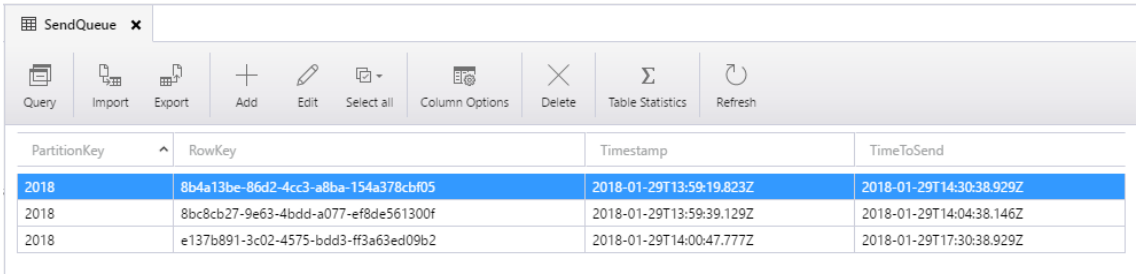

Kuva 3. Key Vaultin hallintanäkymä Azure Portalissa. Kuvassa näkyvät salaisuudet ovat vaihdettavissa helposti ja nopeasti.

### 3.1.7 Azure Web Job

Azure Web Job on ominaisuus, joka sisältyy kaikkiin Azure App Service -palveluihin. Web Jobin idea on, että sillä on helppo suorittaa kevyitä ohjelmia tai skriptejä. Web Jobeja on kahdentyyppisiä: jatkuva (continuous) sekä laukaistu (triggered). Jatkuva Web Job pyörii koko ajan, ja se on laukaistu vain silloin, kun ajastin tai joku muu sen laukaisee. PdfLetterService hyödyntää Web Jobeja ajastetun lähettämisen hoitamiseen. Web Job on asetettu pyörimään läpi kerran minuutissa. Tällöin se tekee kyselyn Azure Table Storagen tauluun ”SendQueue”, joka sisältää tiedon kirjeen id:stä ja kellonajasta jolloin kirjeen pitää lähteä. Mikäli taulussa sijaitseva tieto ”TimeToSend” on vähemmän kuin nykyinen kellonaika, Web Job tekee pyynnön PdfLetterServicelle kirjeen lähettämisestä.

### 3.1.8 Azure Table Storage

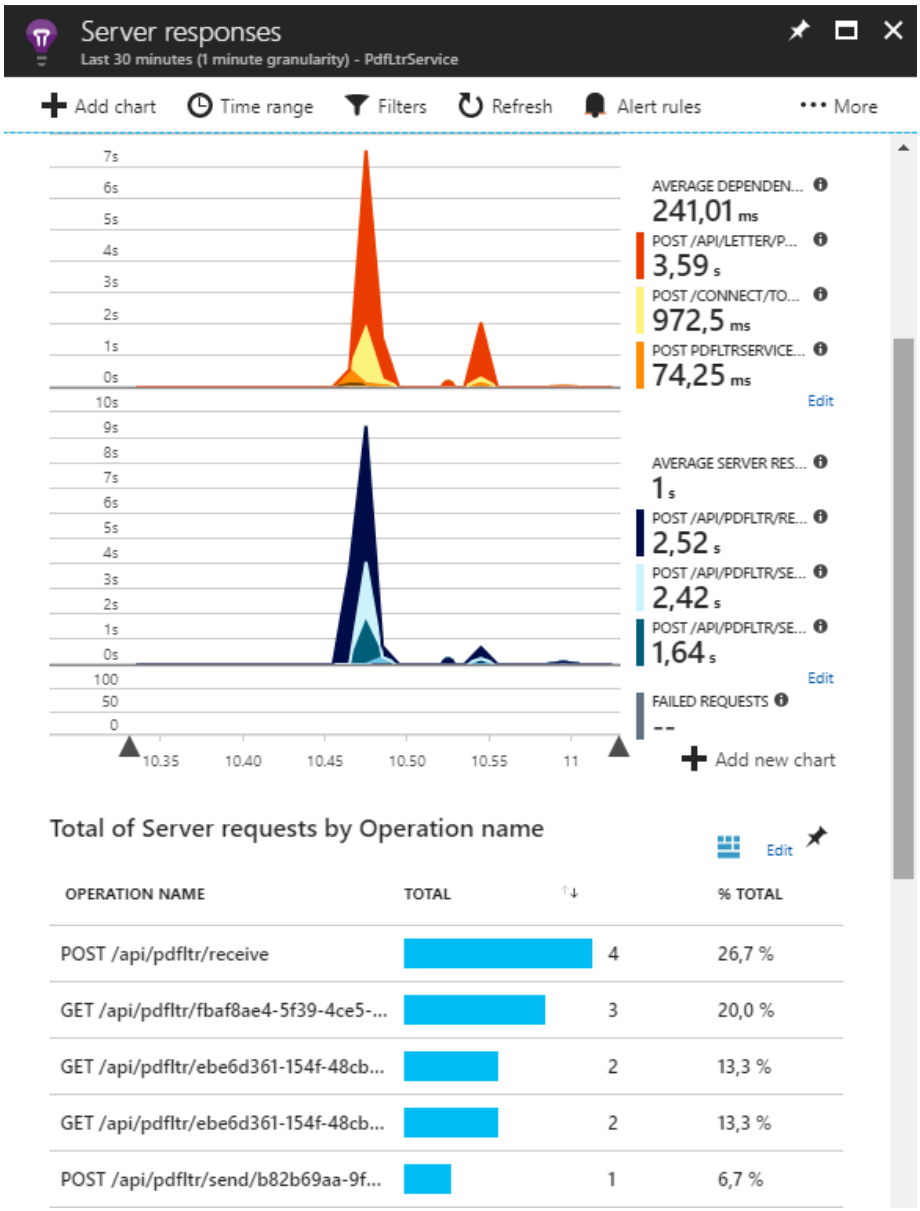
Azure Table Storage on NoSQL-tiedonsäilytyspalvelu, joka käyttää avain-arvopareja. Azure Table Storageen on mahdollista tallentaa petatavujen edestä tietoa [8]. Tiedon hakeminen Table Storagesta on nopeaa ja tehokasta. PdfLetterService käyttää Table Storagea lähetys jonon ylläpitämiseen. SendQueue-nimisessä taulussa on tieto kirjeen id:stä sekä kellonajasta, jolloin kirje lähetetään. Kaikki lähtevät kirjeet kulkevat lähetysjonon kautta.




Kuva 4. Azure Table Storagen taulu ”SendQueue” katsottuna Azure Storage Explorer-ohjelmalla. Nähtävissä on kolme kirje-id:tä ja niille lähetysajat.

### 3.1.9 Azure Application Insights

Azure Application Insights on diagnostiikkatyökalu, joka kerää tietoa sovelluksen toiminnasta. Application Insights asennetaan projektiin mukaan, ja sen jälkeen se pyörii taustalla automaattisesti ja lähettää kerätyt diagnostiikat pilveen. Kerättyä dataa voi katsella ja analysoida helposti Azure Portalin kautta.




Kuva 5. Azure Application Insights -näkymä Azure Portalissa.

PdfLetterServicen suorituskykyä voidaan seurata reaaliajassa. Kuvassa 5 näkyy esimerkki tiedon esitystavasta. Tietoa voi suodattaa ja katsella erilaisin kaavioin. On myös mahdollista katsella yksittäisiä pyyntöjä tarkemmin ja nähdä tarkasti, mihin pyynnön suorituksessa on kulunut aikaa. Esimerkiksi kaikki ulkoisiin palveluihin, kuten Azure Cosmos DB:hen ja Azure Blob Storageen kulunut aika on eriteltävissä selkeästi.

## 3.2 Luokat ja datamodelit

### 3.2.1 Saapuva data

PdfLetterService ottaa vastaan datan JSON-muodossa. JSON-muodon etuja ovat mm. helppo luettavuus ihmissilmällä sekä muodon yleinen tunnettavuus. JSON-muotoinen data on helppo serialisoida suoraan C#-luokasta muodostetuksi objektiksi.

```
{
  "pdf": {
    "fileName": "string",
    "attachmentDataBase64": "string",
    "compressionMethod": "string",
    "pdfPassword": "string"
  },
  "dateToSendUTC": "string",
  "externalId": "string",
  "letterClass": 0,
  "isTesting": 0,
  "letterName": "string",
  "notifyAddress": "string",
  "receiverInfo": {
    "receiverName1": "string",
    "receiverAddress1": "string",
    "receiverZipCode": "string",
    "receiverCity": "string",
    "receiverCountry": "string",
    "receiverEmail": "string"
  },
  "senderInfo": {
    "senderName": "string",
  },
  "sendImmediatly": 0,
}
```

Esimerkkikoodi 1. Karsittu malli vastaanotettavasta JSON-muotoisesta datasta.

PfLetterServicessä ASP.NET tekee serialisoinnin automaattisesti saapuvasta HttpRequestMessage-pyynnöstä, kun saapuva parametri on merkitty [FromBody]-atribuutilla. Saapuvalle datalle tehdään ns. ”Model Validation”, jolla tarkastetaan datan oikea muoto ja että siellä on vaadituissa kentissä arvoja. ASP.NET-viitekehys osaa tehdä tämän automaattisesti yksinkertaisille kentille, esim kaikki int- ja string-kentät. Mikäli data on vääränmuotoista tai vaadittuja kenttiä puuttuu, ModelState.IsValid- muuttuja saa arvon false. Tämän jälkeen API:n kutsujalle palautetaan HTTP-virhekoodi 400 ja tieto siitä, mitkä kentät olivat virheellisiä.

Esimerkkikoodi 1:ssä pakollisia kenttiä ovat ”externalId”, ”receiverName1” ja ”receiverCountry”. Jos jokin näistä puuttuu, pyyntö hylätään. Esimerkkikoodi 2:ssa näkyvän funktion ensimmäisellä suoritus rivillä tarkistetaan, onko ”Model Validation” mennyt läpi.

```
[HttpPost]
public async Task<IHttpActionResult> ReceiveLetterAndQueueForSend([From-
Body]LetterDataInWithPdf letter)
{
      if (!ModelState.IsValid)
            return BadRequest(ModelState);

      PdfLtrServiceActionResponse<LetterModelDb> response = await reposi-
      tory.SaveNewLetterAsync(letter);

      if (response.operationSuccess)
            return Ok(response.GetEntity());
      else
            return BadRequest(response.message);
}
```

Esimerkkikoodi 2. PdfLetterServicen kontrollerin tärkein funktio ”ReceiveLetterAndQueueForSend”. Funktio vastaanottaa ja käsittelee saapuvan datan. Saapuva parametri on tyyppiä LetterDataInWithPdf ja se on merkitty [FromBody]-atribuutilla.

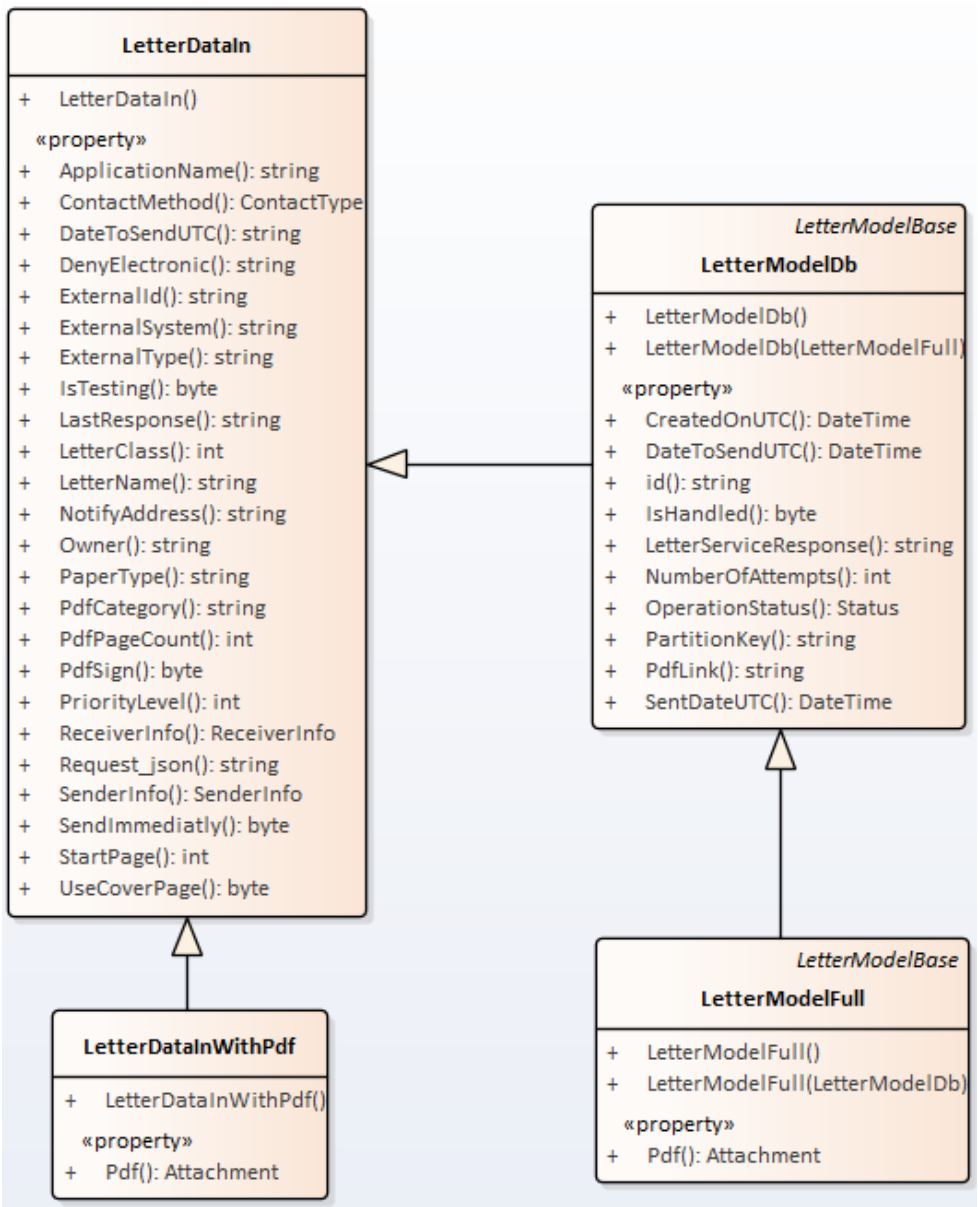
Onnistuneen modelin validoinnin jälkeen saapuvan pyynnön data on LetterDataInWithPdf-luokasta muodostetussa objektissa. Sen jälkeen välitetään data seuraavalle funktiolle, ja se voi olla varma, että data on oikeassa muodossa eikä ylimääräisiä tarkistuksia tarvitse tehdä.

### 3.2.2 Tallennettava data

Luokkia suunnitellessa pitää miettiä ja valita partitioavain (partition key). Hyvällä partitioavaimella varmistetaan datan nopea ja tehokas haku isosta tietokannasta. Azure Cosmos DB vaatii partitioavaimen käyttöä tietokannoissa, joiden maksimikoko on asetettu isommaksi kuin 10 gigatavua. Partitioavaimen saa itse valita kaikkien tallennettavien kenttien joukosta. Azure Cosmos DB:hen tallennettavan LetterModelDb-tyyppisen dokumentin partitioavaimelle päätettiin tehdä kokonaan oma kenttänsä. Partitioavain muodostetaan kirjeen id:n kahdesta ensimmäisestä heksaluvusta. Esimerkiksi jos kirjeen id on ”a94bb284-2af0-4231-abc6-399183025799”, sen partitioavaimeksi valitaan ”a9”.

Partitioavaimeksi haluttiin valita sellainen tieto, jonka voi saada helposti selville kirjeen id:stä. Tämä mahdollistaa sen, että kirjeiden hakeminen onnistuu pelkällä id-luvulla, josta PdfLetterService osaa näppärästi parsia kaksi ensimmäistä lukua. Koska

partitioavain muodostuu kahdesta heksaluvusta, on kaiken kaikkiaan mahdollista olla 256 erilaista partitioavainta. Lähetettyjen kirjeiden säilytysikä on välillä 1-2 vuotta, jonka jälkeen ne poistetaan. Näin ollen 256:n partitioavaimen tulisi riittää hyvin tarvittavalle kirjemäärälle ja taata nopea haku tietokannasta.




Kuva 6. PdfLetterServicen tiedontallennukseen käyttämät luokat. Nähtävissä ylin luokka LetterDataIn sekä sen perivät aliluokat.

Kuvasta 6 on nähtävissä PdfLetterServicen tärkeimmät dataluokat. Ylin luokka on LetterDataIn, joka sisältää kaiken mahdollisen vastaanotettavan datan, paitsi PDF-tiedoston. PDF-tiedostoa ei ole ylimmässä luokassa, koska on mahdollista ottaa vastaan kirjeitä, joissa ei ole PDF-tiedostoa mukana. Näihin kirjeisiin PDF-tiedosto voidaan lisätä myöhemmin. Ylimmästä luokasta on olemassa lähes identtinen luokka LetterDataInWithPdf, joka sisältää myös PDF-datan.

LetterModelDb-luokka perii LetterDataIn-luokan. LetterModelDb sisältää kaiken tiedon, joka tallennetaan Azure Cosmos DB:hen. LetterModelDb ei sisällä ollenkaan PDF:n dataa, vain ainoastaan string-tyyppisen kentän ”PdfLink”, joka sisältää linkin Azure Blob Storageen tallennettuun PDF-tiedostoon.

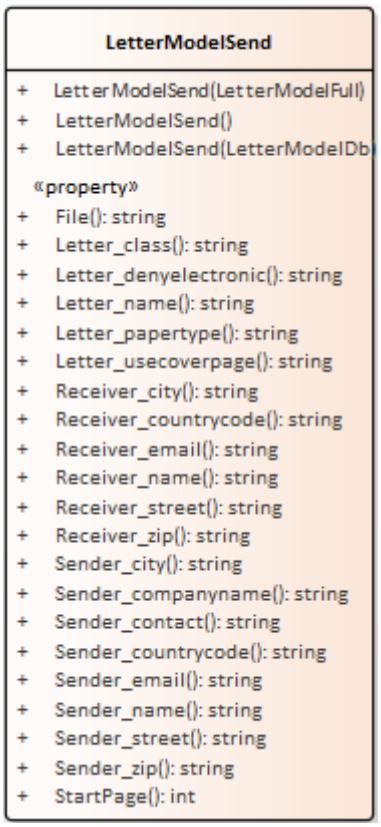
Azure Blob Storagen linkit ovat muotoa muodoltaan normaaleja url- linkkejä, esimerkiksi seuraavanlaisia: https://pdfletterservice.blob.core.windows.net/pdf/37fe94b9-055b-4252-a4d3-4bc4cdc6f2fe. Linkkien avaaminen onnistuu, mikäli käyttäjällä on hallussaan SAS (Shared Access Signature) tai Azure Blob Storagen connection string.

Koska LetterModelDb ei sisällä ollenkaan PDF-tyyppistä dataa, on kirjeiden hakeminen tietokannasta nopeaa ja tehokasta, eikä turhia resursseja mene hukkaan. Mikäli halutaan hakea vain tietyn kirjeen lähetysaika ja vastaanottaja, on hyvä, ettei jopa monen megatavun kokoista PDF-tyyppistä dataa tarvitse turhaan kuljettaa pyynnön mukana. Tämä on tärkeää, koska Azure Cosmos DB:n käytön kustannukset nousevat käytettyjen Ru/s (resource units per second) mukaan.

LetterModelFull-luokka perii kaiken LetterModelDb-luokalta. Tämä luokka sisältää PDF-tyyppisen datan kokonaisena ja on tarkoitettu kyselyitä varten, jossa halutaan tallennetun kirjeen kaikki data, mukaan lukien myös PDF base64 stringinä kokonaan ulos.

### 3.2.3 Lähtevä data

PdfLetterService lähettää kirjeen ulkoiselle palvelulle. Ulkoinen palvelu vastaanottaa kirjeen tiedot JSON-muodossa. Tätä varten tehtiin luokka, jossa ovat kaikki vastaanotettavat tiedot muuttujina. Muuttujat nimettiin juuri samalla tavalla, kuin ulkoinen palvelu haluaa ne. Näin ollen luokka LetterModelSend voidaan serialisoida suoraan JSON:ksi, ja se kelpaa sellaisenaan ulkoiselle palvelulle.




Kuva 7. PdfLetterServicen käyttämä LetterModelSend-luokka.

Kuvassa 7 näkyvän LetterModelSend-luokan muuttujat on nimetty eri tavalla verrattuna muihin PdfLetterServicen luokkiin. LetterModelSend-luokassa käytetään alaviivaa sanojen välissä. Tämä johtuu ulkoisen palvelun erilaisesta tavasta ottaa data vastaan.

## 3.3 Palvelun ohjelmointi

### 3.3.1 Asynkroninen vs. synkroninen toteutus

Asynkroninen ohjelmointi on tehokas tapa saada etenkin Web API:hin lisää skaalattavuutta ja tehoa. Asynkronisessa ohjelmoinnissa on tarkoituksena välttää säikeen tukkiminen silloin, kun odotetaan vastausta muulta palvelulta [9]. Esimerkiksi jos synkronisesti toteutettu funktio hakee tietokannasta yhden rivin dataa, ja haku kestää jostain syystä kauan, tällöin yksi säie odottaa toimettomana koko tämän ajan turhaan vieden arvokasta suoritusaikaa muilta pyynnöiltä. Käytettävissä olevia prosessoriytimiä on yleensä hyvin rajallinen määrä, joten yksi tukkiva säie saattaa viedä useita kymmeniä

prosentteja käytettävissä olevasta suoritustehosta pelkästään odottamalla tietokannasta tulevaa vastausta.

Asynkroninen ohjelmointi tarjoaa ratkaisun edellä mainittuun ongelmaan. Ratkaisu on vieläpä yhtä nopea ja selkeä toteuttaa kuin synkroninen versio. Idea on yksinkertaistettuna seuraavan lainen: kun kutsutaan aikaa vievää funktiota, sen sijaan että jäätäisiin odottamaan paikalleen tekemättä mitään, palautetaankin nykyinen säie säievarantoon (thread pool), josta se on käytettävissä muita tehtäviä varten.

```
public async Task<bool> UpdateLetterAsync(LetterModelDb updatedLetter)
{
      var result = await documentClient.ReplaceDocumentAsync(UriFactory.Cre-
      ateDocumentUri(databaseId, collectionId, updatedLet-ter.id),updatedLet-
      ter);
      return result.StatusCode == System.Net.HttpStatusCode.OK;
}
```

Esimerkkikoodi 3. Asynkroninen funktio, joka kutsuu DocumentClientin asynkronista funktiota, joka päivittää Azure Cosmos DB:ssä olevan dokumentin. Tässä tapauksessa dokumentti on PdfLetterService-kirje.

Esimerkkikoodi 3:ssa funktion ensimmäisellä suoritusrivillä kutsutaan DocumentClient-luokasta muodostettua objektia. DocumentClient-luokka on osa ”Azure SDK for .NET” -pakettia ja se mahdollistaa tietokantakyselyiden ajamisen kantaa vasten. DocumentClientin funktio ”ReplaceDocumentAsync” on asynkroninen funktio, jonka tunnistaa yleisestä käytännöstä nimetä asynkronisia funktioita päätteellä Async. Asynkronisen funktion palauttama arvo on aina tyyppiä Task tai Task<TResult>.

Oleellisin asia asynkronisessa funktiokutsussa tapahtuu await-operaattorin kohdalla. Esimerkkikoodi 3:n funktiokutsussa await documentClient.ReplaceDocumentAsync(), await-operaattorin käyttö tarkoittaa, että kyseisen funktion suoritus aloitetaan ja sen jälkeen nykyinen säie palautetaan säievarantoon. Tämän ansiosta säie ei ole tukittuna odottamassa vastausta itsestä riippumattomalta palvelulta, jolla vastaaminen saattaa kestää kauan. Nyt prosessi voi käyttää säiettä muiden operaatioiden suorittamiseen. Järjestelmä tunnistaa automaattisesti, kun funktiokutsun documentClient.ReplaceDocumentAsync() palauttama Task on ”Completed”-tilassa. Sitten järjestelmä nappaa säievarannosta yhden vapaan säikeen ja asettaa sen suorittamaan loput funktion await-operaattorin jälkeisistä riveistä.

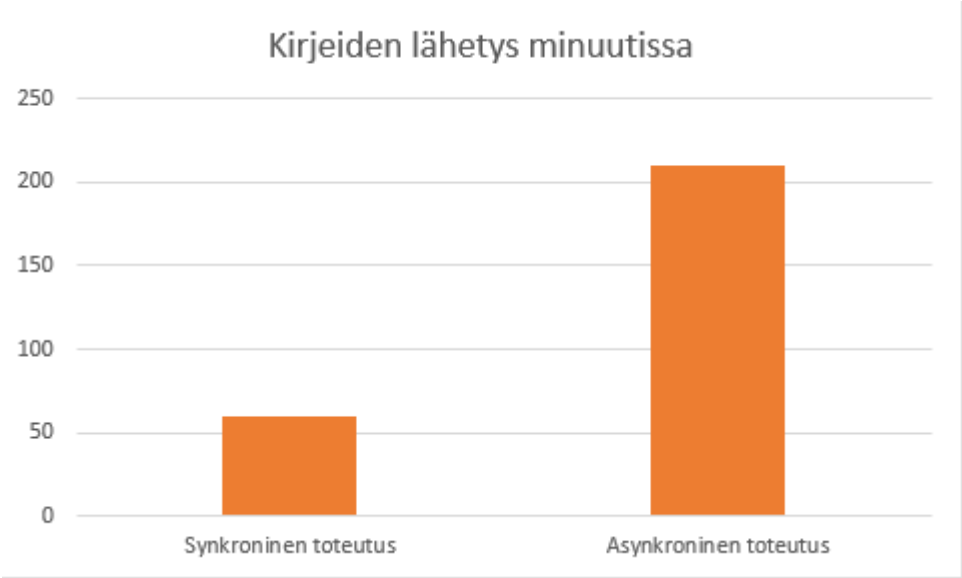
PdfLetterServicen ensimmäisessä toteutuksessa kaikki funktiot olivat synkronisia. Ohjelma toimi hyvin ja sulavasti normaaleissa testisimulaatioissa. Ongelmaksi kuitenkin muodostui suoritusajan riippuvuus siitä, kauanko ulkoisella palvelulla kestää vastata PdfLetterServicen lähettämään pyyntöön.

Testituloksia tarkasteltaessa on huomioitava, että kaikki testit suoritettiin ohjelman kehityksessä käytetyllä kannettavalla tietokoneella. Käytetyn laitteiston CPU oli Intel Core i5-7300U @ 2.60GHz. Mikäli testit olisi suoritettu PdfLetterServicen ollessa hostattuna Azuren App Serviceen, tulokset olisivat olleet vielä rankemmin asynkronisen toteutuksen kannalla. Syy tähän on se, että PdfLetterServicen asynkroninen toteutus olisi mahdollista skaalata App Servicessä todella suuriin lukemiin. Esimerkiksi sen voisi laittaa pyörimään 100 eri instanssissa, joista jokaisella instanssilla on 4 ydintä käytössä [10]. Näin ollen asynkroninen toteutus voisi olla helposti jopa 100 kertaa nopeampi pilvessä kuin kannettavalla tietokoneella testattuna.

Edellä mainittu skaalaaminen maksaa enemmän riippuen käytettävästä ”App Service planista”. Kun PdfLetterService on valmis ja tuotannossa, se tulee pyörimään Standard-tason App Service planissa, jossa on käytettävissä 10 instanssia, jotka jaetaan kaikkien planiin liitettyjen sovellusten kesken.

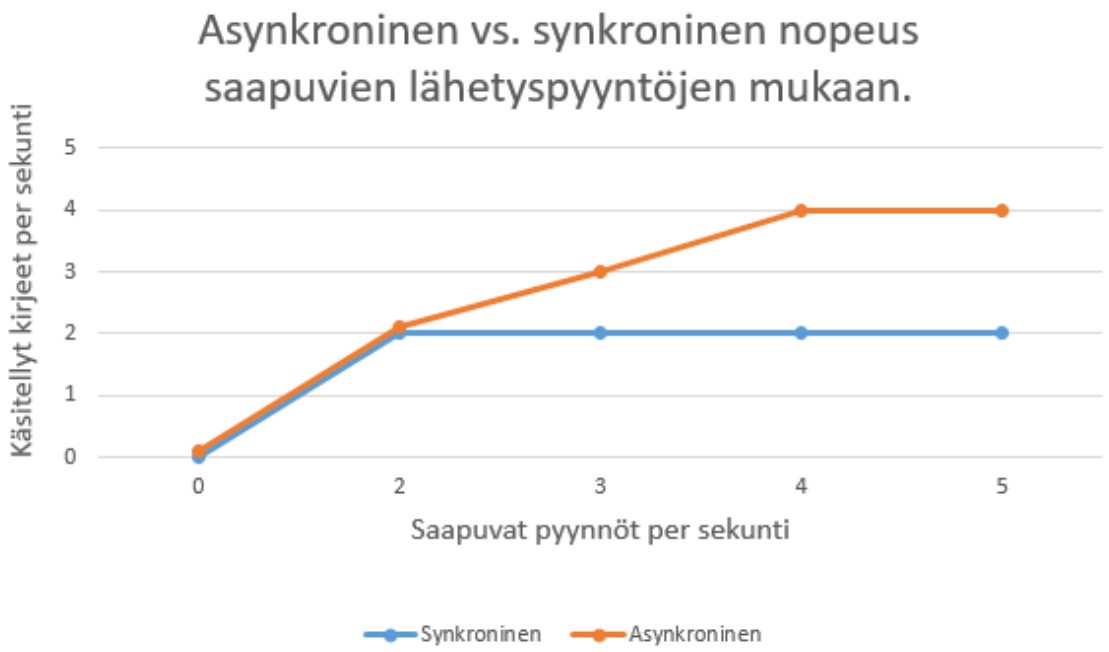
Testisimulaatioissa ulkoisen palvelun vastausajaksi asetettiin 2 s. Tämän jälkeen huomattiin, että PdfLetterServicen toiminta hidastui suoraan verrannollisesti ulkoisen palvelun vastausaikaan.

Ratkaisu tähän ongelmaan oli muuttaa kaikki PdfLetterServicen käyttämät funktiot asynkronisiksi. Testisimulaatiot ajettiin uudestaan sekä 2 s:n vastausajalla että 0,2 s:n vastausajalla. Siinä vaiheessa huomattiin, että vastausajan muutos ei enää aiheuttanut PdfLetterServicen toimintanopeuteen muutoksia. Kirjeiden lähetysnopeus ennen asynkronisia muutoksia ja 2 s:n vastausajalla oli noin 50-70 minuutissa. Asynkronisten muutosten jälkeen lähetysnopeus 2 s:n vastausajalla oli noin 180-240 kirjettä minuutissa.




Kuva 8. Kirjeiden maksimilähetysnopeus tilanteessa, jossa lähetyspyyntöjä tulee paljon, esimerkiksi 300 minuutissa.

Lähetysnopeuden muutoksesta on huomattavissa asynkronisen ohjelmoinnin selkeä hyöty tietyntyyppisissä tilanteissa. Asynkroninen ohjelmointi ei kuitenkaan kaikissa tapauksissa nopeuta sovelluksen toimintaa näin merkittävästi. Esimerkiksi yhden yksittäisen kirjeen lähettäminen synkronisilla tai asynkronisilla funktiokutsuilla vie saman verran aikaa. Asynkronisuuden hyöty korostuukin tilanteissa, joissa tulee paljon pyyntöjä suoritettaviksi samaan aikaan. Tällöin pyyntöjä pystytään ajamaan rinnakkain enemmän verrattuna synkroniseen toteutukseen. PdfLetterServicelle tällainen tilanne on muutaman kerran kuukaudessa lähetettävät massakirjeet, joita lähtee yhden yön aikana vähintään 6000 kappaletta.




Kuva 9. Kaaviosta nähdään, että kun saapuvia pyyntöjä tulee 2/s, molemmat sekä asynkroninen että synkroninen toteutus ovat yhtä nopeita lähettämään kirjeen eteenpäin. Kun taas pyyntöjä tulee 4/s, asynkroninen on selvästi nopeampi.

Asynkronisuus tuo ohjelman skaalautuvuuteen erittäin suuren lisän. Koska asynkronisesti toteutetussa ohjelmassa säikeet tekevät koko ajan työtä, on ohjelman todellista suorituskykyä helppo mitata. Suorituskyvystä voidaan seurata esimerkiksi CPU:n (Central Processing Unit) käyttöastetta. Azuren App Servicessä hostatun ohjelman skaalaa voidaan muuttaa manuaalisesti tai se voidaan asettaa muuttumaan automaattisesti tietyn parametrin mukaan. Yksi parametreista on CPU:n käyttöaste, joka voidaan asettaa esimerkiksi maksimiarvolle 75 %, jonka jälkeen Azure lisää automaattisesti enemmän prosessoriytimiä ohjelman käytettäväksi. Vastaavasti voidaan asettaa myös minimiarvo CPU:n käyttöasteelle, esimerkiksi 25 %, jolloin Azure vähentää automaattisesti prosessoriytimien määrää. Tässä kohdassa asynkronisesti toteutetun ohjelman skaalautuvuus näkyy parhaiten. Mikäli ohjelma olisi toteutettu synkronisesti, ei ytimien lisääminen olisi läheskään yhtä tehokasta, koska osa säikeistä olisi koko ajan odottamassa vastausta ulkopuoliselta palvelulta.

PdfLetterServiceä voidaan helposti skaalata ylöspäin massakirjelähetysten aikaan, jotta kirjeiden lähetys sujuu jouhevasti. Normaaliolosuhteissa lähetysmäärä on n. 200 kirjettä

per päivä. Tällöin skaalautuvuudesta ja asynkronisuudesta ei ole PdfLetterServicen toimintanopeudelle suurta merkitystä. Tämä voidaan todeta kuvasta 6, jossa verrataan asynkronisuuden ja synkronisuuden eroja pienellä määrällä vastaanotettavia pyyntöjä.

### 3.3.2 Autentikaatio ja palvelun käytön rajoittaminen

Sovellus, joka on yhdistetty internettiin ilman tietoturvan asiallista toteutusta, on riskialtis väärinkäytölle. Olisi erittäin ikävä tilanne, jos joku ulkopuolinen pahantekijä pääsisi käsiksi PdfLetterServicen kirjeen lähetystoimintoihin. Näin ollen pahantekijä voisi yrittää lähettää kirjeitä mielin määrin mihin tahansa osoitteisiin tuhansittain. Mikäli PdfLetterService ei autentikoisi saapuvia pyyntöjä millään tavalla, pahantekijä pystyisi kutsumaan palvelua tietämällä vain palvelun url-osoitteen. Azuren App Serviceen hostatuttujen palveluiden url-osoitteet ovat muotoa: https://apiname.azurewebsites.net. Tällaisessa url-osoitteessa ’apiname’-kohdalle tulee oman palvelun nimi, jotenka osoitteita ei ole vaikea arvailla.

PdfLetterService autentikoi kaikki saapuvat HTTP-pyynnöt. Autentikaatioon käytetään ”Mutual Certificate Authentication”-menetelmää, sekä IP-osoitteiden rajausta. PdfLetterService tarkistaa kaikilta pyynnöiltä sertifikaatin. Sertifikaatin pitää vastata konfiguraatioasetuksissa olevaa asiakkaan (client) sertifikaattia. Sertifikaatti ladataan konfiguraatioasetuksiin Azuren Key Vaultista. Näin ollen sertifikaatin vaihtaminen on helppoa, koska ohjelman koodiin ei tarvitse tehdä muutoksia sertifikaattia vaihdettaessa. Tarkistus on toteutettu lisäämällä ClientCertificateAuthentication-väliohjelmisto (middleware) Owin (Open Web Interface for .NET) pipelineen.

```
public void Configuration(IAppBuilder appBuilder)
{
      ...
      appBuilder.UseClientCertificateAuthentication(new DefaultClientCertifi-
      cateValidator());
      ...
}
```

Esimerkkikoodi 4. Startup.cs-tiedostossa on Configuration-funktio, joka suoritetaan kerran ohjelman käynnistyessä. Se on oikea paikka lisätä Owin pipelineen tarvittavat väliohjelmistot. Tässä on lisätty ClientCertificateAuthentication-väliohjelmisto. Muunlaisissa ohjelmissa se voisi olla korvattu esimerkiksi FacebookAuthentication-väliohjelmistolla.

Esimerkkikoodissa 4 esitelty ClientCertificateAuthentication-väliohjelmisto pitää huolen siitä, että kaikki pyynnöt, joissa ei ole oikeaa sertifikaattia mukana, hylätään. Oikea sertifikaatti on vain Azure API Managementin tiedossa, joten saapuvaa pyyntöä, jossa on oikea sertifikaatti mukana, voidaan pitää jo pitkälti luotettavana.

### 3.3.3 Kirjeen lähetysprosessi

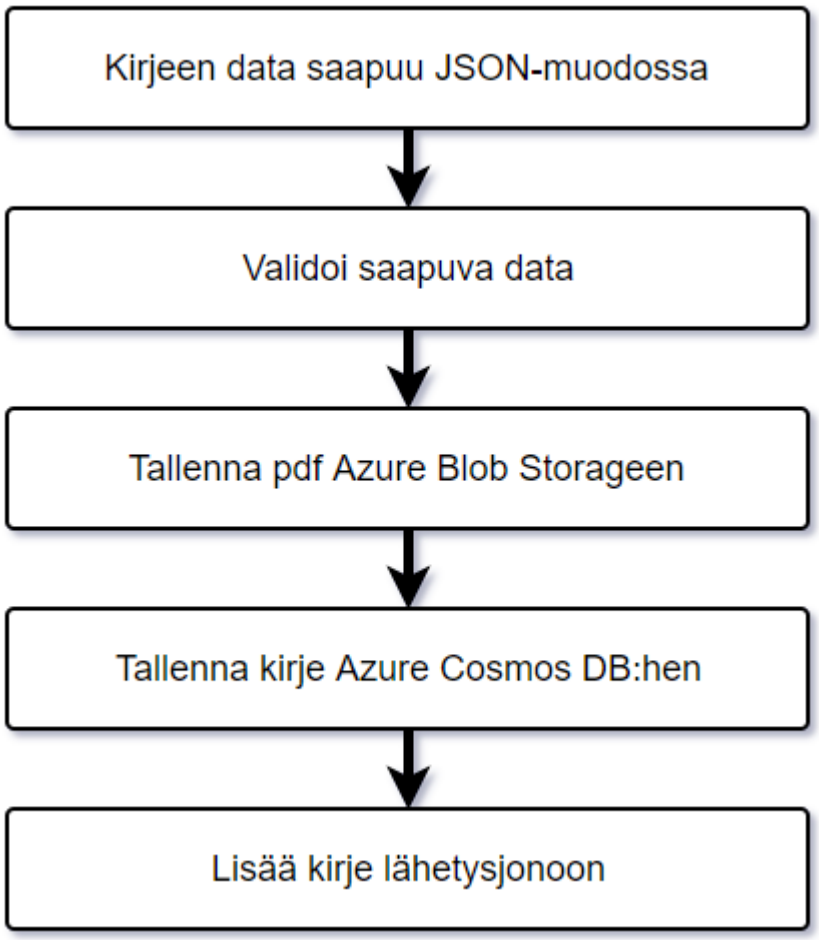
PdfLetterServicen kirjeen lähetysprosessi jakautuu kahteen osaan. Ensimmäinen on kirjeen vastaanottaminen. Toinen on kirjeen lähettäminen eteenpäin silloin, kun siihen saadaan käsky.

Lähetysprosessin ensimmäinen vaihe alkaa siitä, kun vastaanotettaessa uusi kirje, se ensin tarkistetaan väärien arvojen varalta. Mikäli tarkistus menee läpi, kirjeen PDF tallennetaan Azure Blob Storageen. Blob Storagelta saatu kyseisen PDF:n linkki tallennetaan Azure CosmosDB:hen kaiken muun datan kanssa.

Tämän jälkeen kirjedatasta luetaan parametrin ”dateToSendUTC” arvo, joka tarkoittaa aikaa, jolloin kirje pitää lähettää. Mikäli ajankohtaa ei ole määritetty, PdfLetterService asettaa kirjeen lähetysajan konfiguraatiotiedostossa olevan minuuttimäärän verran eteenpäin nykyajasta. Sen jälkeen, kun lähetysaika on asetettu oikein, kirjeen id sekä lähetysaika tallennetaan Azure Table Storagessa sijaitsevaan tauluun SendQueue. Kirjeen id on muodostunut Azure Cosmos DB:n tallennusvaiheessa, ja se on GUID (Globally Unique Identifier) -tyyppinen arvo.

Seuraavaksi PdfLetterService ei tee kirjeelle mitään, vaan se odottaa erillistä lähetyspyyntöä, joka luetaan sisään samalla tavalla REST-rajapinnan kautta kuin muukin data. Lähetyspyynnön lähettämisestä PdfLetterServicelle vastaa Azure Web Job. Tämä Web Job on asetettu suorittamaan ”ScanQueueAndSend- funktio kerran minuutissa. Käytännössä tämä tarkoittaa sitä, että aina suorittuessaan funktio tekee kyselyn SendQueue-tauluun ehdolla, onko ”TimeToSend” pienempi kuin nykyinen kellonaika. Sen jälkeen saadut tulokset käydään silmukalla läpi ja tuloksen id:t lähetetään PdfLetterServicelle REST-rajapinnan kautta. Funktio katkaisee silmukan, kun silmukkaa on pyöritty konfiguraatiotiedostossa määritetyn ”maxNumberOfSendRequestsPerMinute”-arvon verran. Näin ollen PdfLetterServicen kirjeen lähetysnopeuteen voidaan vaikuttaa helposti muuttamalla kyseistä arvoa.

Ajastetun lähetyksen yksi hyödyistä on PdfLetterServiceen kohdistuvan samanaikaisen kuormituksen väheneminen. Esimerkiksi 6000 kirjeen erä voidaan vastaanottaa kello 20:00 ja lähetys aloittaa vasta 00:00, jolloin saapuvat kirjeet eivät enää hidasta alkavaa lähetysprosessia.




Kuva 10. Lähetysprosessin ensimmäinen vaihe, jossa kirje otetaan vastaan. Sen jälkeen jäädään odottamaan kirjeen lähetyskäskyä.

Lähetysprosessin toinen vaihe alkaa siitä, kun saatuaan kirjeen lähetyspyynnön, PdfLetterService tekee vielä muutaman tarkistuksen ennen kirjeen lähettämistä eteenpäin. Ensimmäiseksi tarkistetaan, onko kirjeessä ”IsTesting”-arvo päällä. Jos näin on, kirjeen vastaanottajan tiedot korvataan konfiguraatiotiedostossa määritetyillä arvoilla. Näin varmistetaan se, että testikirjeitä ei lähde vahingossa ulkopuolisille ihmisille.

```
public enum Status
    {
        NotSet,
        QueuedForSend,
        SuccessfullySent,
        Failed,
        WaitingForLetterService,
        WaitingForData,
        WaitingForSend
    }
```

Esimerkkikoodi 5. Status-enum, jolla pidetään kirjaa siitä, mikä on Azure Cosmos DB:ssä olevan kirjeen lähetyksen tila.

Seuraavaksi kirjeestä tarkistetaan ”OperationStatus”-muuttujan arvo. Tärkeimpiä tarkistuksista ovat mm. vertaaminen Status-enumin arvoon ”SuccessfullySent” ja ”WaitingForLetterService”, jolla varmistetaan, että jo lähetettyä kirjettä ei lähetetä missään tapauksessa uudestaan.

Tarkistusten mentyä läpi kirjeen ”OperationStatus” asetetaan ”WaitingForLetterService”-tilaan. Tämä tehdään siksi, että jos pyyntö ulkoiseen palveluun kestää yli minuutin, Web Job löytää tämän saman kirjeen lähetysjonosta ja antaa käskyn lähettää kirje uudelleen. Silloin olisi mahdollista, että sama kirje lähtisi kahteen kertaan.

Seuraavaksi PdfLetterService alkaa rakentaa ulos lähtevää HTTP-pyyntöä. HTTP- pyynnön lähettämiseen käytetään NuGet-pakettia RestSharp. Pyynnön rakennuksessa on kaksi vaihetta; ensimmäinen on kirjeen datan lisääminen pyynnön bodyyn JSON-muotoisena.

Toinen vaihe on autentikaatio tiedon lisääminen pyynnön headeriin. Lisätty autentikaatio tieto määärytyy sen mukaan, mitä konfiguraatiotiedostossa on asetettu ulkoisen palvelun autentikaatiomenetelmäksi. Tätä työtä toteutettaessa ulkoinen palvelu käytti ”Token based authentication”-menetelmää, mutta joskus myöhemmin se tulee olemaan Azure API Managementin takana, jonka jälkeen token-autentikaatiota ei enää tarvita. Työn tilaajan toiveesta tehtiin helpoksi se, että on helppo vaihdella näiden kahden, Token based sekä API Management autentikaation välillä. Tämä onnistuu lukemalla arvot konfiguraatiotiedostosta, jolloin itse ohjelman koodiin ei tarvitse tehdä muutoksia.

Seuraavaksi, kun lähtevä HTTP-pyyntö on valmis, PdfLetterService lähettää kirjeen ulkoiselle palvelulle asynkronisesti. Vastauksen saavuttua siitä tarkistetaan HttpStatusCode. Mikäli tämä koodi on ”OK”, pyyntö on mennyt onnistuneesti läpi, ja kirje on lähtenyt eteenpäin.

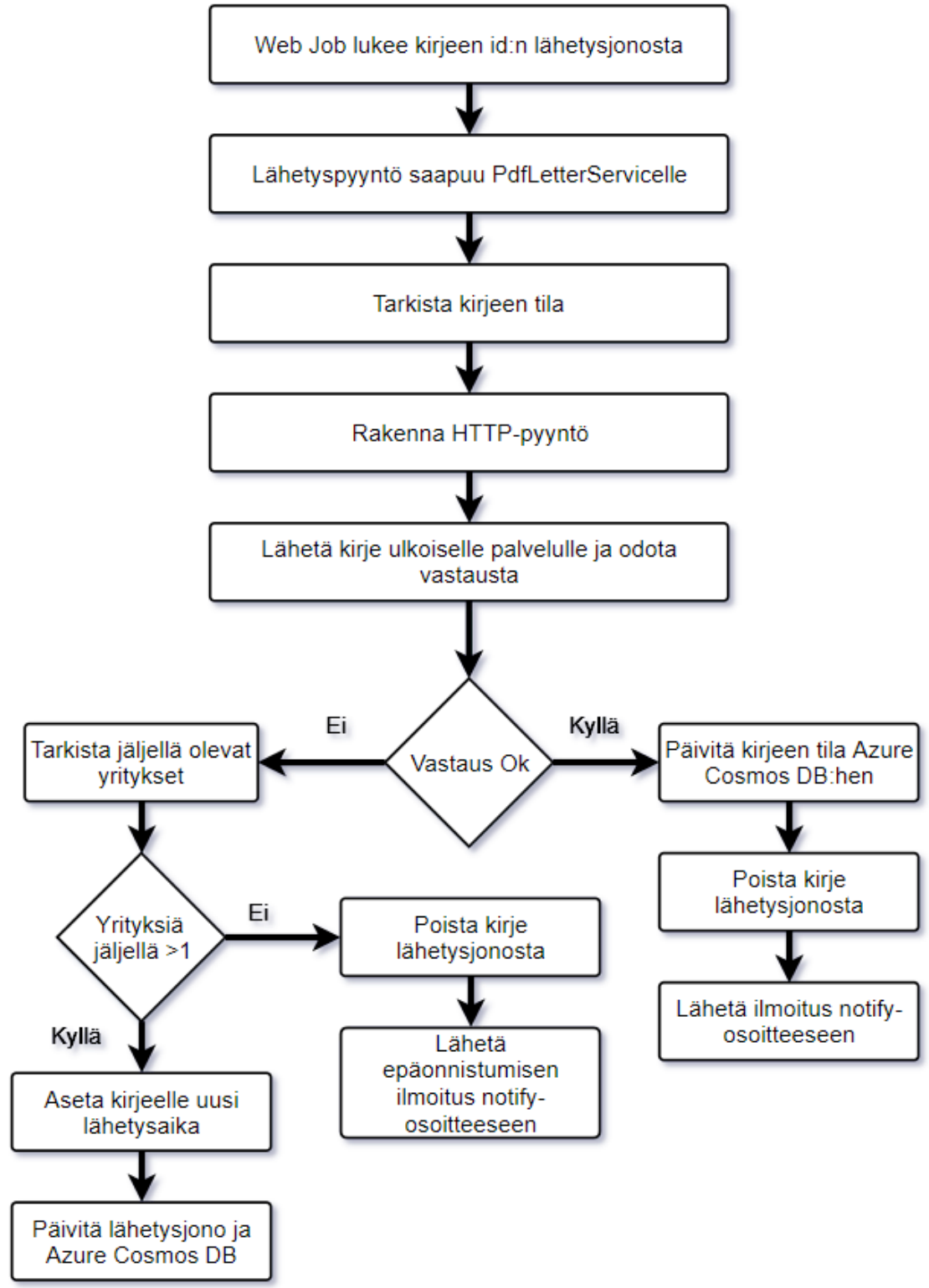
Onnistuneen pyynnön jälkeen kirje poistetaan ”SendQueue”-taulusta, jolloin Web Job ei enää pyydä lähettämään tätä kirjettä. Myös kirjeen ”OperationStatus” asetetaan arvoon ”SuccessfullySent”. Ulkoinen palvelu sisällyttää onnistuneeseen vastaukseen id- tunnuksen, jonka PdfLetterService ottaa talteen kirjeen ”LetterServiceResponse”-kenttään. Tämän jälkeen lähetetään vielä ilmoitus onnistuneesta lähetyksestä osoitteeseen, joka on vastaanotettu muun kirjedatan joukossa. Se, joka on kirjeen alun perin lähettänyt PdfLetterServicelle, saa tässä vaiheessa tiedon kirjeen onnistuneesta lähetyksestä.

```
{
      "SendSuccessfull": "bool",
      "Message": "string"
}
```

Esimerkkikoodi 6. Yksinkertainen JSON, joka lähetetään kirjeen mukana mahdollisesti toimitettuun notify-osoitteeseen.

Mikäli ulkoiselta palvelulta saatu vastaus on muuta kuin ”OK”, PdfLetterService kasvattaa kirjeen ”NumberOfAttempts”-arvoa yhdellä ja kirjaa ”LetterServiceResponse”-kenttään vastauksen kokonaisuudessaan. Konfiguraatiotiedostossa on määritetty, mikä on suurin arvo, joka ”NumberOfAttempts” voi olla. Mikäli se on lisäyksen jälkeen suurempi kuin sallittu, kirjeen ”OperationStatus” asetetaan tilaan ”Failed” ja kirje poistetaan ”SendQueue”-taulusta. Tämän tarkoituksena on välttää tilanne, jossa PdfLetterService yrittäisi turhaan lähettää ulkoiselle palvelulle kirjettä, jota se ei koskaan hyväksy.

Silloin kun ”NumberOfAttempts” on pienempi kuin suurin sallittu arvo, kirjeen ”OperationStatus” vaihdetaan ”WaitingForSend”-tilaan ja kirjeen ”DateToSendUTC”-arvoksi asetetaan nykyinen aika plus konfiguraatiotiedostossa määritelty minuuttimäärä. Näin epäonnistunut lähetys laitetaan ikään kuin jäähylle määritetyksi ajaksi, jonka jälkeen sen lähettämistä yritetään uudelleen, kun Web Job löytää kirjeen id:n ”SendQueue”-taulusta.




Kuva 11. Kirjeen lähetysprosessin toinen vaihe

### 3.3.4 Virheettömän sekä jatkuvan toiminnan varmistaminen

On tärkeää, että PdfLetterService toimii koko ajan virheettömästi. Mikäli toiminnan aikana kuitenkin tapahtuu virheitä, on ne käsiteltävä asianmukaisesti. Virheistä on myös hyvä kirjata ylös kaikki mahdolliset tiedot, jotta voidaan tutkia, mistä virhe johtuu, ja myöhemmin korjata se.

Kaikki PdfLetterServicen tärkeimmät funktiot on ”suojattu” try-catch-lausekkeilla. Tärkeimmillä funktioilla tarkoitetaan funktioita, jotka aloittavat muiden funktioiden kutsumisen, eli ne jotka jäävät kutsupinon alimmaiseksi. Näin ollen, mikäli missä tahansa vaiheessa tapahtuu virhe, se saadaan napattua ja hoidettua asianmukaisesti. Try-catch-lauseita on myös muuallakin koodissa, mikäli tilanne sitä vaatii. Yksi tällainen tilanne on esimerkiksi, kun kutsutaan funktiota, joka hakee kirjeen id:n perusteella Azure Cosmos DB:stä käyttäen apuna Azure SDK:n DocumentClient-luokkaa. Jos kirjeen haku epäonnistuu, DocumentClient palauttaa virheen, joka on tyyppiä DocumentClientException.

```
public async Task<LetterModelDb> GetLetterFromDbByIdAsync(string letterId)
{
      try
      {
            var partitionKey = letterId.Substring(0, 2);
            LetterModelDb foundLetter;
            var result = await documentClient.ReadDocumentAsync<LetterModelDb>(
                  UriFactory.CreateDocumentUri(databaseId, collectionId, let-
                  terId), new RequestOptions { PartitionKey = new Parti-
                  tionKey(partitionKey) });

            foundLetter = (LetterModelDb)(dynamic)result;
            return foundLetter;
      }
      catch (DocumentClientException ex)
      {
            if(ex.StatusCode == (System.Net.HttpStatusCode)429)
            {
                  Log.Error("Db error 429TooManyRequests." + ex);
            }
            return null;
      }
      catch (Exception ex)
      {
            Log.Error(ex);
            return null;
      }
}
```

Esimerkkikoodi 7. Funktio joka hakee Azure Cosmos DB:stä kirjeen parametrina saadulla id:llä. Mahdollinen DocumentClientin palauttaman virhe käsitellään erillisessä catch-osiossa.

Tutkimalla tarkemmin DocumentClientExceptionia saadaan selville mistä, syystä haku epäonnistui. DocumentClientException sisältää tiedon siitä, minkä HTTP- virhekoodin Azure Cosmos DB on palauttanut. Tästä virhekoodista voidaan tulkita, miksi haku epäonnistui. Syitä voivat olla esimerkiksi se, että kirjettä ei löydy, tai Azure Cosmos DB:hen asetettu suoritustehon (throughput) maksimiraja on saavutettu. Koska virhe käsitellään jo tässä vaiheessa sen sijaan että, se menisi kutsupinossa alimmalle kutsujalle asti, saadaan selkeämpi rakenne kaikkien virheiden käsittelyyn. Esimerkkikoodi 7:n tapauksessa virheen tulkitsemisen jälkeen palautetaan arvo null funktion kutsujalle.

Mahdollisten epäonnistuneiden toimintojen havaitsemiseksi suurin osa funktiokutsuista palauttaa ”PdfLtrServiceActionResponse<T>”-tyyppisen objektin. Tämä luokka sisältää tiedon kutsun onnistumisesta ja objektin, esimerkiksi LetterModelDb-tyyppisen, sekä viestin.

```
public class PdfLtrServiceActionResponse <T>
{
      public bool operationSuccess { get; } = false;
      private T entity;
      public string message { get; } = "";
}
```

Esimerkkikoodi 8. PdfLtrServiceActionResponse-luokka, jonka suurin osa PdfLetterServicen funktioista palauttaa kutsujalleen. Esimerkkikoodista on jätetty pois konstruktorit sekä muutama funktio.

Funktiokutsun jälkeen, kun tarkastellaan saatua Esimerkkikoodi 8:n kaltaista objektia, on helppo havaita, onko kutsu onnistunut. Tarvitsee vain lukea booleanin ”OperationSuccess” arvo. Mikäli arvo on epätosi, aloitetaan palaaminen kutsupinossa taaksepäin, kunnes saavutaan kontrollerin funktioon, joka on alun perin tehnyt ensimmäisen kutsun. Esimerkkikoodi 2 näyttää yhden kontrollerifunktion, jonka palauttama vastaus riippuu ”OperationSuccess”-muuttujan arvosta.

```
public async Task<PdfLtrServiceActionResponse<string>> SendLetterToLetter-
ServiceAsync(string id)
{
      var letterData = await CheckLetterStatus(id);

      if (!letterData.operationSuccess)
            return new PdfLtrServiceActionResponse<string>(false, "", letter-
            Data.message);

      var pdfData = await database.GetPdfFromStorageAsync(letterData.GetEn-
      tity().PdfLink.Substring(letterData.GetEntity().PdfLink.LastIndexOf('/')
      + 1));

      if (!pdfData.operationSuccess)
      {
            letterData.GetEntity().OperationStatus = Status.WaitingForData;

            await database.UpdateLetterAsync(letterData.GetEntity());
            await database.DeleteLetterFromSendQueueAsync(id, letterData.GetEn-
            tity().ExternalId);
            return new PdfLtrServiceActionResponse<string>(false, "", "Failed to
            send. Letter has invalid pdf data");
      }


      return await Send(letterData.GetEntity(), pdfData.GetEntity());
}
```

Esimerkkikoodi 9. Osa funktiosta, joka aloittaa kirjeen lähetyksen. Ensimmäiseksi funktio tarkistaa kirjeen tilan kutsumalla CheckLetterStatus()-funktiota. Saatu vastaus on PdfLtrServiceActionResponse-tyyppinen ja siitä katsotaan muuttujan operationSuccess-arvo. Sama tehdään, kun ladataan kirjeen PDF-tiedosto Azure Blob Storagesta. Mikäli kumpikaan operaatio ei epäonnistu, voidaan jatkaa kirjeen lähetykseen.

Virhe- ja lokitiedon automaattiseen kirjaamiseen PdfLetterService käyttää NuGet-pakettia log4net sekä log4net.Appender.Azure. log4net on työkalu, joka auttaa lokitietojen keräämisessä ja tekee siitä helppoa. log4netiä voi käyttää kutsumalla missä tahansa luokassa funktiota log4net.LogManager.GetLogger()-funktiota, joka palauttaa Logger-tyyppisen objektin. Sen jälkeen tälle Logger-tyyppiselle objektille voidaan tehdä funktiokutsu, jolla se tallentaa annetun tekstin lokiin.

log4net sisältää erilaisia loki tasoja. Erilaiset tasot auttavat jaottelemaan tärkeät lokiviestit omaan kategoriaan ja vähemmän tärkeät omaansa. Tasoja ovat All, Debug, Info, Warn, Error, Fatal, Off. Esimerkiksi ohjelmassa tapahtuvat virheet yleensä lokitetaan Error-tasolle. Tieto kirjeen käsittelyyn ottamisesta lokitetaan Info-tasolle. Kun lokeista etsitään tietoa, esimerkiksi virheitä, tulokset voidaan suodattaa lokitason perusteella. Tällöin etsintää eivät häiritse kymmenettuhannet Info-tason viestit.

```
Log.Error(ex);
Log.Info("Test information");
Log.Debug("Debug message..");
```

Esimerkkikoodi 10. Log on GetLogger()-funktion palauttama Logger-tyyppinen objekti. Esimerkkikoodissa on kutsuttu kolmea eri loki tasoa, Error, Info, Debug.

Konfiguraatiotiedostossa on määriteltävissä paikka, johon Logger-objekti kirjoittaa lokimerkinnät. log4net käyttää ”appendereja” lokien kirjoittamiseen. Appenderilla tarkoitetaan palikkaa, joka konfiguroidaan kirjaamaan tietyn tasoiset lokit tiettyyn paikkaan. PdfLetterService käyttää lokien kirjaamiseen kahta appenderia. Ensimmäinen on ”RollingFileAppender”, joka kirjoittaa lokitason ”All” tekstitiedostoon ”PdfLtrService.log”, joka sijaitsee ohjelman suorituskansiossa. Toinen on ”AzureTableAppender”. AzureTableAppender mahdollistaa lokien kirjaamisen Azure Table Storagen tauluun. Myös tämä appender on asetettu kirjaamaan kaikki lokitasot.

Appenderit voidaan konfiguroida toimimaan erilaisilla parametreilla. Muutamat parametrit ovat tärkeitä ohjelman suorituskyvyn ja lokien luettavuuden kannalta. Esimerkiksi RollingFileAppenderin parametri ”maximumFileSize” on hyvä asettaa sopivan kokoiseksi, jotta lokitietoa ei turhaan säilötä kymmeniä tai jopa satoja megatavuja. Toinen parametri, joka on syytä ottaa huomioon, on AzureTableAppenderin ”bufferSize”. Tämä parametri määrää käytettävän puskurin koon. Mikäli puskurin koko olisi hyvin pieni, esimerkiksi vain 1, tällöin AzureTableAppender lähettäisi jokaisen lokimerkinnän erikseen Azure Table Storagen lokitauluun. Tämä hidastaisi ohjelman toimintaa, koska yhteys jouduttaisiin avaamaan jokaisella lähetyksellä uudestaan. Vaikkakin ohjelman hidastuminen tässä tapauksessa olisi vain muutaman kymmenen millisekunnin luokkaa, järkevämpi tapa on asettaa puskurin koko tarpeeksi isoksi. Esimerkiksi puskurin koon ollessa 50, AzureTableAppender joutuu avaamaan yhteyden AzureTableStorageen vain kerran jokaista 50 lokimerkintää kohden.

Esimerkkikoodi 7:ssä on nähtävillä Logger-objektin käyttö try-catch-osion catch-lauseissa, joissa kirjataan tapahtunut virhe tasoon Error. Tällöin kaikki appenderit, jotka on asetettu kirjaamaan taso Error ylös, tekevät sen.

PdfLetterService kirjaa tietoa myös muuten kuin pelkällä log4net-työkalulla. Halutaan pitää kirjaa asioista, kuten kaikki lähteneet kirjeet tai kaikki epäonnistuneet kirjeet. Onnistuneille sekä epäonnistuneille -kirjeille on omat taulunsa Azure Table Storagessa. Näitä kahta kirjattavaa tietoa varten on rakennettu yksinkertainen funktio.

```
private async Task WriteToTableLog(LetterModelDb letter, Status status)
{
      if (status == Status.SuccessfullySent)
            await WriteToSentLettersLog(letter);
      else if (status == Status.Failed)
            await WriteToFailedLettersLog(letter);
      return;
}
```

Esimerkkikoodi 11. Funktio, joka vastaanottaa kirjeen datan sekä statuksen. Saapuva status määrittää sen, mihin lokiin kirjeen tiedot kirjoitetaan.

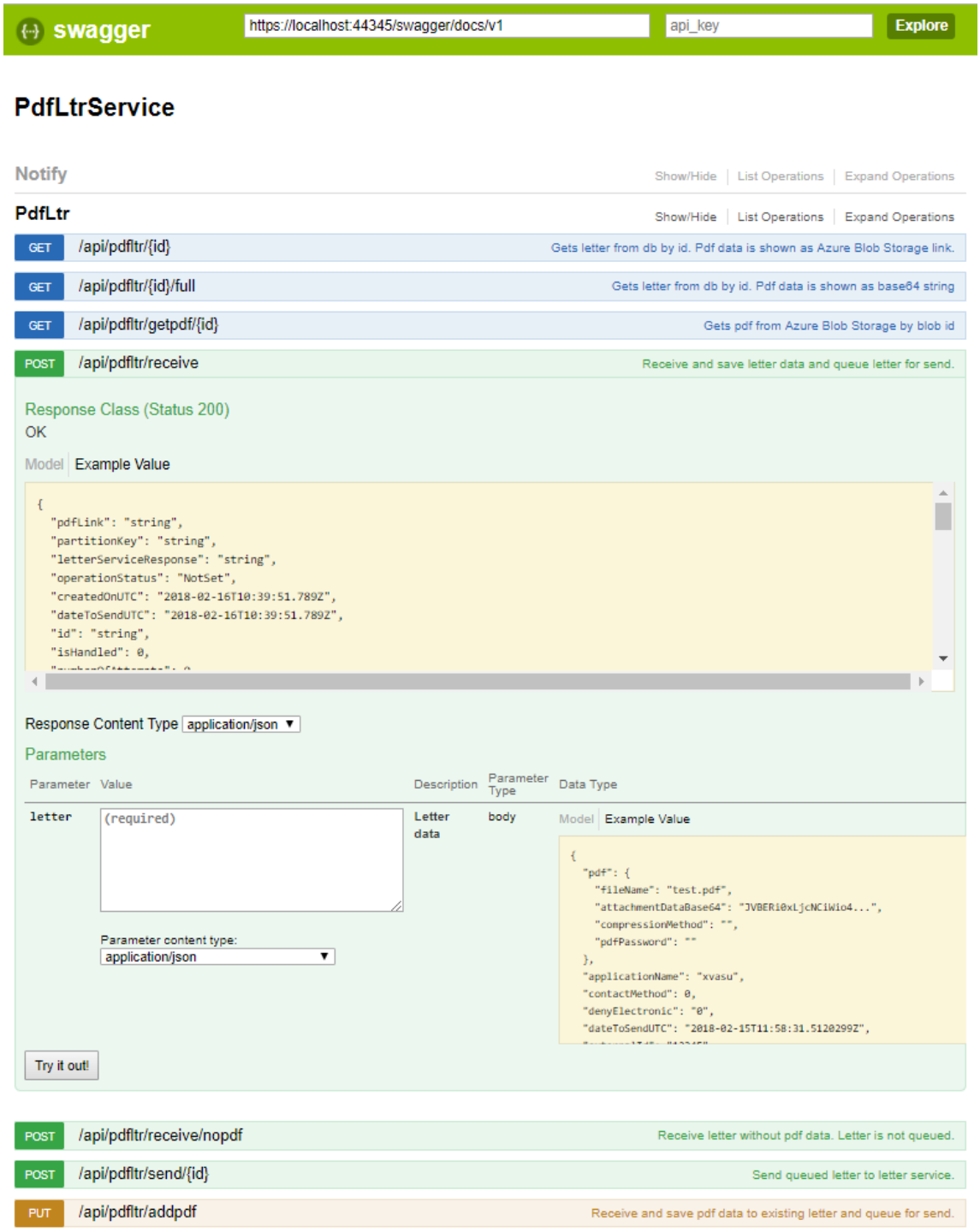
Esimerkkikoodi 11:sta kaltaista funktiota voidaan kutsua PdfLetterServicessä mistä tahansa aina silloin, kun halutaan kirjoittaa kirjeen statuksen muutos lokiin. Funktio tarvitsee vain status tiedon ja päättelee sen mukaan oikean loki funktion, jota kutsua. Funktiota on myös helppo laajentaa tulevaisuudessa, mikäli muita statuksia halutaan kirjata erilaisiin lokeihin.

## 3.4 Testaus

Testaus on tärkeä tehtävä, jota on hyvä tehdä läpi ohjelman kehityksen. PdfLetterServicen testaus hoidettiin käyttämällä useaa erilaista työkalua, jotka mahdollistavat API:n testaamisen erilaisin tavoin. Testaamista tehtiin aina muutosten jälkeen, jotta voitiin olla varmoja, että ohjelma toimii niin kuin pitääkin.

PdfLetterServicelle tehtiin oma TestClient-niminen ohjelma. Se on yksinkertainen konsolisovellus (console application) ja sillä on mahdollista testata PdfLetterServicen kaikkia rajapinnan komentoja. TestClientissä on mahdollista konfiguroida lukumäärä, jonka verran se lähettää kirjeitä PdfLetterServicelle. Tällä testattiin PdfLetterServicen kirjeiden vastaanottokykyä ja edelleen lähetystä. Luvussa 3.3.1 esitellyt kirjeiden lähetysnopeudet on kokeiltu TestClientillä.

Swagger on työkalu, joka on tarkoitettu mm. API:n dokumentointiin ja testaukseen. Swaggerin avulla testaaminen on nopeaa ja visuaalisesti helppoa ilman, että tarvitsee itse koodata mitään.



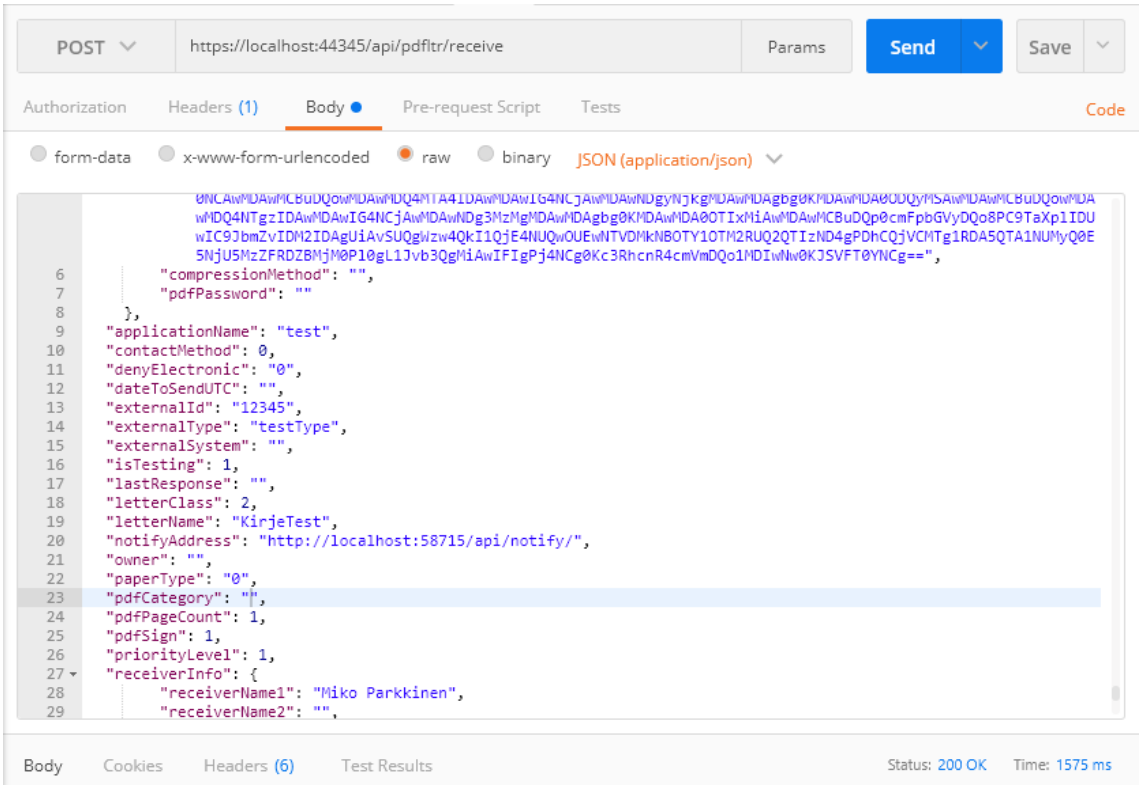

Kuva 12. PdfLetterServicen Swagger-näkymä verkkoselaimella katseltuna. Kuvaa on rajattu näyttämään vain tärkeimmät toiminnot.

Kuvassa 12 näkyvällä Swaggerin testisivulla on mahdollista testata kaikkia PdfLetterServicen rajapinnan funktioita. Funktiot ovat selkeästi eroteltuina eri väreillä sen mukaan, mitä HTTP-verbiä kukin funktio käyttää. Kuvassa on laajennettuna ”/api/pdfltr/receive”-

funktio. Swagger näyttää, minkälaisen resurssin funktio palauttaa, mikäli kutsu on onnistunut. Nähtävillä on myös malli resurssidatasta, jonka funktio ottaa vastaan. Kaikki mallit voi itse määritellä Swaggerin konfiguraatioasetuksista ohjelmakoodissa. Swaggerillla on mahdollista generoida ”OpenApi Specification”-dokumentti PdfLetterServicestä. OpenApi Specification -dokumentti kuvaa kaiken rajapinnan toiminnallisuuden yhdessä dokumentissa [11]. Dokumentista löytyy kaikki funktioiden palauttamista arvoista, nimistä, malleista yms. Tämän OpenApi Specification -dokumentin avulla kuka tai mikä tahansa osaa kutsua PdfLetterServicen funktioita ja odottaa niistä tietynlaista vastausta. Dokumenttia hyödynnettiin, kun PdfLetterService lisättiin Azure Api Managementin taakse.

Azure API Management hyväksyy OpenApi Specification -tyyppisen dokumentin, kun ollaan lisäämässä uutta API:a, johon ohjataan pyyntöjä. OpenApi Specification sisältää miltei kaiken informaation, mitä Azure API Managementin tarvitsee tietää API:sta, jotta siihen voidaan ohjata pyyntöjä.

Swaggerin lisäksi toinen tärkeä sovellus, jota käytettiin PdfLetterServicen testaamisessa oli Postman. Postman on sovellus, jolla on mahdollista tehdä laajempaa API:n testaamista kuin Swaggerilla. Postmanilla voi rakentaa HTTP-pyynnön kokonaan itse. Esimerkiksi auktorisointi (authorization) headerin lisääminen pyyntöön ei onnistu Swaggerilla ilman itse koodattua lisäosaa. Näin ollen välillä oli hyvä vaihdella Postmanin sekä Swaggerin välillä ja hyödyntää molemmista parhaimmat ominaisuudet.




Kuva 13. Rajattu näkymä Postman-sovelluksesta. Näkyvillä JSON, joka sisältää kirjedatan. Tässä tapauksessa se on lähetetty osoitteeseen https://localhost:44345/api/pdfltr/receive käyttäen HTTP-verbiä POST.

Postmanin yksi hienoimmistä ominaisuuksista on lähetettyjen pyyntöjen tallentaminen. Postman tallentaa automaattisesti kaikki lähetetyt pyynnöt. Esimerkiksi kuvassa 13 näkyvä pitkä JSON-tiedosto on kätevää olla Postmanissa tallessa sen sijaan, että sitä joutuisi aina copy-pastettamaan erillisestä muistiosta tms. Myös valmiina oleva url-osoite sekä pyynnön headerit nopeuttavat testitapauksia.

# 4 Lopputulos

Työn tarkoituksena oli toteuttaa palvelu, joka vastaa yrityksen E-kirjeiden lähettämisestä ja korvaa vanhan käytössä olleen palvelun. Lisäksi toteutettavaan palveluun tuli lisätä mahdollisuus kirjeiden ajastettuun lähettämiseen.

Työn tuloksena saatiin aikaan onnistunut projekti. PdfLetterService integroitiin yrityksen järjestelmiin, ja se toimii tuotannossa päivittäisessä käytössä. Työn kestoksi oli arvioitu

5 viikkoa ja tuntimääräksi viikkoa kohden 37,5 tuntia. Työ saatiin päätökseen sovitun aikataulun puitteissa. Arvioituun aikaan ei sisältynyt tämän raportin kirjoittamista.

PdfLetterService helpottaa yrityksen E-kirjeiden lähetysprosessia. Nyt lähtevistä kirjeistä pystyy ennen lähetystä katsomaan kirjeen ulkonäön ja tekemään mahdolliset korjaukset. Aikaisemmin tämä oli hankalampaa, koska tekstistä muodostettiin kirje vasta postin päässä. PDF-muotoisten E-kirjeiden myötä kirjeen ulkoasua voidaan muuttaa ilman kustannuksia. Aiemmin postilla olevan kirjeen mallin muuttaminen maksoi satoja euroja.

PdfLetterService toimii pilvipalveluna Azuressa, joten E-kirjeiden lähettäminen onnistuu mistä tahansa asiakasohjelmasta entisen yhden ohjelman sijaan. Asynkroninen toteutus yhdistettynä pilvipalveluiden skaalattavuuteen takaa tulevaisuudessa PdfLetterServicen toiminnan myös isommalla kuormalla.

## Lähteet


1 Rakli Ry. 2016. Kiinteistönomistajat Suomessa. Verkkoaineisto. http://www.rakli.fi/tietoa-kiinteistoalasta/kiinteistoalan-yhteiskunnallinen-merkitys/faktaa-kiinteistoalasta/suurimmat-kiinteistosijoittajat.html Luettu 19.2.2018.

2 Phadke, Akshay. 2016. Introduction to ASP.NET Web API. Verkkoaineisto. http://www.c-sharpcorner.com/article/introduction-to-asp-net-web-api/ Luettu 14.2.2018.

3 Microsoft Azure. 2018. API Management. Verkkoaineisto. https://azure.microsoft.com/en-us/services/api-management/ Luettu 5.2.2018.

4 Microsoft Azure. 2016. SLA for App Service. Verkkoaineisto. https://azure.microsoft.com/en-us/support/legal/sla/app-service/v1_4/ Luettu 19.2.2018.

5 Microsoft Azure. 2017. SLA for Azure Cosmos DB. Verkkoaineisto. https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/v1_1/ Luettu 19.2.2018.

6 Microsoft Azure. 2018. Blob Storage. Verkkoaineisto. https://azure.microsoft.com/en-us/services/storage/blobs/ Luettu 19.2.2018.

7 Microsoft Azure. 2018. Key Vault. Verkkoaineisto. https://azure.microsoft.com/en-gb/services/key-vault/ Luettu 19.2.2018.

8 Microsoft Azure 2018. Azure Table Storage. Verkkoaineisto https://azure.microsoft.com/en-gb/services/storage/tables/ Luettu 19.2.2018.

9 Microsoft Azure. 2016. Asynchronous Programming. Verkkoaineisto. https://docs.microsoft.com/en-us/dotnet/csharp/async/ Luettu 19.2.2018.

10 Microsoft Azure. 2018. App Service Pricing. Verkkoaineisto. https://azure.microsoft.com/en-us/pricing/details/app-service/ Luettu 19.2.2018.

11 Swagger Docs. What is OpenAPI?. Verkkoaineisto. https://swagger.io/docs/specification/about/ Luettu 15.2.2018