

IoT-järjestelmän laite- ja asiakkuudenhallinta- sovellus

LAB-ammattikorkeakoulu

Liiketalous, Digitradenomi

2020

Tuomas Käyhty, Arto Lindgren

Tiivistelmä

Tekijä(t) Käyhty, Tuomas Lindgren, Arto	Julkaisun laji Opinnäytetyö, AMK	Valmistumisaika 2020
	Sivumäärä 43	
Työn nimi IoT-järjestelmän laite- ja asiakkuudenhallintasovellus		
Tutkinto Liiketalouden tradenomi		
Ohjaavan opettajan nimi, titteli ja organisaatio Jouni Könönen, Lehtori, Konetekniikka		
Toimeksiantajan nimi, titteli ja organisaatio		
<p>Tiivistelmä</p> <p>Opinnäytetyön tavoitteena oli suunnitella ja toteuttaa asiakkuuksien- ja laitehallintaan tarkoitettu fullstack web-sovellus tilaajan olemassa olevalle IoT-järjestelmälle.</p> <p>Opinnäytetyön toteutuksessa hyödynnettiin ketteriä ohjelmistokehitysmenetelmiä, jotka koskettivat ohjelmistotuotannon kaikkia osa-alueita työskentelytavoista aina suunnitteluun ja toteutukseen asti.</p> <p>Opinnäytetyönä toteutetun sovelluskokonaisuuden tietokantana oli Azure SQL-tietokanta, joka toimi Azuren pilviympäristössä. Tietokantaa operoitiin SQL Server Management Studiolla. Palvelinpuoli toteutettiin .NET Core Web API projektina Microsoftin Visual Studiolla ja käyttöliittymä Angular-sovelluksena JetBrains Webstorm ohjelmointiympäristössä. Myös palvelin ja käyttöliittymä julkaistiin Azureen.</p> <p>Sovelluskokonaisuuden toteutus onnistui todella hyvin ja kaikki tarvittavat vaatimukset täyttyivät. Sovellus on asiakasyrityksen käytössä. Opinnäytetyö kasvatti tekijöiden ammattitaitoa ja he hyötyvät projektissa opituista asioista tulevalla urallaan.</p>		
Asiasanat .net core, fullstack, angular		

Abstract

Author(s) Käyhty, Tuomas Lindgren, Arto	Type of Publication Thesis, UAS Number of Pages 43	Published 2020
Title of Publication Management software solution for controlling an IoT- system		
Name of Degree Bachelor of Business Administration		
Name, title and organization of the supervising teacher Jouni Könönen, Senior Lecturer, Mechanical engineering		
Name, title and organization of the client		
<p>Abstract</p> <p>The purpose of this thesis was to design and develop a managing software for managing customer information and device parameters for the customer's existing IoT-system.</p> <p>The software development was done using Agile practices, which affected the whole development process including designing, planning, development, and the day-to-day work.</p> <p>The produced software solution was done as a Full Stack project. Database used in the project was an Azure SQL database. The server-side software was done as a .NET Core Web API application, developed with Microsoft Visual Studio. The client-side was an Angular application, developed using the JetBrains WebStorm IDE. Both the server and client applications were published to Azure.</p> <p>The resulting software solution of the development project was a success, and all the necessary requirements were met, and the software went in production. The development project increased the professional skills of its makers and will benefit them further in their careers.</p>		
Keywords .net core, fullstack, angular		

Sisällys

1	Johdanto.....	1
1.1	Toimeksiantaja	1
1.2	Ongelman esittely.....	1
1.3	Sovelluksen rakenne	1
1.4	Opinnäytetyön rakenne.....	1
1.5	Työnjako.....	2
2	Prosessi ja menetelmät.....	3
2.1	Agile	3
2.2	Järjestelmän vaatimusten määrittely	6
2.2.1	Vaatimusmäärittely	6
2.2.2	Confluence	6
2.3	Käyttöliittymäsuunnittelu	6
2.3.1	UI/UX suunnittelu.....	7
2.3.2	Suunnittelutyökalu Adobe XD	7
2.4	API-rajapinta ja sen suunnittelu	8
2.4.1	REST-rajapinta ja suunnittelun hyviä käytäntöjä	8
2.4.2	OpenAPI 3.0.....	10
2.4.3	Postman	10
2.5	Projektinhallinta	10
2.5.1	Scrum	11
2.5.2	Jira	13
3	Arkkitehtuurin kuvaus.....	15
3.1	Järjestelmän arkkitehtuuri	15
3.2	Backend sovelluksen rakenne	16
3.3	Frontend sovelluksen rakenne	17
4	Teknologiat	21
4.1	Relaatiotietokanta	21
4.1.1	SQL Server.....	22
4.1.2	SQL Server Management Studio	22
4.2	Backend sovellus.....	22
4.2.1	.NET Core, C#	22
4.2.2	ASP.NET Core Web Application – Web API	22
4.2.3	NuGet.....	23
4.2.4	Microsoft Visual Studio kehitysympäristö	23
4.3	Microsoft Azure.....	23
4.3.1	Azure App Services	23

4.3.2	Azure AD	23
4.3.3	Azure Pipelines.....	24
4.4	IoT-laitehallinta järjestelmät	24
4.5	Frontend sovellus	24
4.5.1	Angular Framework	24
4.5.2	Angular SPA application	24
4.5.3	Node Package Manager	25
4.5.4	JetBrains WebStorm kehitysympäristö.....	25
4.6	Versionhallinta	25
4.6.1	Git.....	26
4.6.2	BitBucket	26
5	Projektin toteutus	27
5.1	Vaatimusmäärittely	27
5.2	UI/UX suunnittelu.....	28
5.3	API suunnittelu	28
5.4	Projektinhallinta	30
5.5	Sovelluskehitys.....	32
6	Pohdinta	36
7	Jatkokehitys.....	38
8	Yhteenveto	39
	Lähteet	40

Käsitteet ja lyhenteet

API	Application Programming Interface: ohjelmointirajapinta, joka määrittelee miten sovellukset kommunikoivat keskenään.
Azure	Microsoftin tarjoama pilvipalvelu, johon voi muun muassa julkaista sovelluksia.
Azure AD	Microsoftin tarjoama identiteettipalvelu Microsoft-tilien hallintaan.
Backend	Sovelluksen palvelinpuoli eli käyttäjän näkymättömissä tapahtuva koodi, kuten tietokantaoperaatiot.
Backlog	Tuotteen kehitysjono, eli projektinhallinnassa käytetty työkalu, johon on lisätty kaikki tuotekehitykseen liittyvät tehtävät.
Epic	Ketterään kehitykseen liittyvä termi, joka kuvaa isompaa kokonaisuutta ohjelmistoprojektista. Epic voidaan jakaa edelleen pienempiin taskeihin.
Frontend	Sovelluksen selainpuoli eli käyttöliittymä.
Fullstack	Sovelluskehityksen lähestymistapa, jossa kehittäjän työnkuvaan kuuluu sovellusjärjestelmän jokainen osa-alue, tietokanta, palvelinpuoli ja käyttöliittymä.
IoT	Internet Of Things: Esineiden internet eli laitteet, joilla on identiteetti ja toimivat internet yhteydellä.
JSON	JavaScript Object Notation: kevyt avoimen standardin tiedostomuoto tiedon välitykseen.
MVVM	Model-View-ViewModel: Arkkitehtuurimalli, joka erottaa käyttöliittymän sovelluslogiikasta.
REST	Representational State Transfer: arkkitehtuurinen malli, jonka avulla määritellään API rajapinta.
RWD	Responsive Web Design: Käyttöliittymän skaalautuminen laitteen ruudun koon mukaan.
Scrum	Ketterään kehitykseen liittyvä projektinhallinnan viitekehys.
SPA	Single Page Application: Yhden sivun sovellus, jossa uudelle sivulle siirryttäessä ei ladata kaikkea tietoa uudelleen.
Sprint	Scrum viitekehitykseen liittyvä aikamääre, joiden aikana työtä tehdään. Ohjelmistoprojekti jakautuu useampaan sprinttiin.

Sprint planning Palaveri, jossa suunnitellaan sprintin aikana tehtävä työ.

Sub Task Työtehtävän alatehtävä, joka kuvaa tarkemmalla käytännön tasolla, mitä työtehtävässä pitää tehdä.

Task Työtehtävä, joka kuvaa mitä halutaan tehdä. Task jakautuu edelleen sub taskeihin.

UCD User Centered Design: Käyttäjäkeskeinen suunnittelu.

UI User Interface Design: Käyttöliittymän suunnittelu.

UX User Experience Design: Käyttökokemuksen suunnittelu.

YAML Ihmisluettava merkintäkieli.

.NET Microsoftin kehittämä ohjelmistokomponenttikirjasto.

1 Johdanto

Tässä opinnäytetyössä tavoitteena on suunnitella ja toteuttaa web-pohjainen hallintasovellus yrityksen olemassa olevalle järjestelmälle. Hallintasovellus on yhdistetty järjestelmä, jolla on mahdollista hallita yrityksen asiakkaita ja yrityksen tarjoamia IoT-laitteita yhdestä sovelluksesta.

Työn käytännöllinen osuus toteutetaan kesätyönä asiakasyrityksessä ja opinnäytetyön kirjallinen osuus koostuu järjestelmän toteutuksen esittelystä.

1.1 Toimeksiantaja

Sisältö on poistettu asiakkaan pyynnöstä.

1.2 Ongelman esittely

Yrityksellä on tarve helppokäyttöiselle järjestelmälle, joka nopeuttaa asiakkuuksien ja laitteiden hallintaa yhdistämällä niiden ylläpidon yhden sovelluksen alle. Uusien asiakkaiden tietojen lisääminen, sekä nykyisten asiakkaiden tietojen päivitys on aikaisemmin tehty osin manuaalisesti muokkaamalla tietokantoja, joka on hidasta ja aikaa vievää. Myös laitteiden hallinta tapahtuu osin manuaalisesti kahdella eri IoT-alustalla, joten keskitetty hallintajärjestelmä helpottaa ja nopeuttaa tiedonhallintaprosesseja.

1.3 Sovelluksen rakenne

Sovelluksen käyttöliittymä on web-pohjainen Angular sovellus, johon kirjaudutaan sisään käyttäen Microsoftin Azure AD käyttäjätunnuksia. Backend sovellus on C# ohjelmointikielellä tehty .NET core Web API (Application Programming Interface), joka keskustelee SQL-tietokannan, IoT-laitteiden hallintajärjestelmien sekä Angular sovelluksen välillä.

1.4 Opinnäytetyön rakenne

Opinnäytetyössä esitellään sovelluksen tuotantoprosessiin liittyvät vaiheet siinä järjestyksessä, jonka mukaan prosessi eteni. Projektin kaikkiin osa-alueisiin liittyvät ketterän ohjelmistokehityksen periaatteet esitellään aluksi. Työn alussa toteutettiin sovelluksen vaatimusmäärittely, suunniteltiin sovelluksen ulkonäköön ja käytettävyyteen liittyvät asiat (UI/UX) ja lopuksi suunniteltiin sovelluksien välistä REST-rajapintaa (Representational State Transfer). Kaikkien suunnitteluun liittyvien prosessien jälkeen alkoi sovellusten kehitys, joka toteutettiin Scrum-viitekehityksen mukaisesti.

Ohjelmistokehitysmenetelmien jälkeen esitetään järjestelmän arkkitehtuuri, palvelinpuolen sovelluksen ja käyttöliittymäsovelluksen rakenne sekä tämän jälkeen toteutuksessa käytetyt teknologiat. Seuraavaksi avataan järjestelmän eri vaiheiden toteutusta käytännössä. Pohdinnassa tutkitaan, miten projekti onnistui kokonaisuudessaan, mitä olisi voinut tehdä toisin, sekä täyttyivätkö vaatimusmäärittelyssä asetetut tavoitteet. Lopuksi mietitään mahdollisia jatkokehitysmahdollisuuksia ja tehdään opinnäytetyön yhteenveto.

1.5 Työnjako

Sovelluskehityksen työnjako jakautuu siten, että toinen opinnäytetyön tekijöistä toteuttaa pääosin frontend-sovelluksen ja toinen puolestaan backend-sovelluksen. Molemmat osallistuvat vaatimusmäärittelyn toteutukseen, UI/UX-suunnitteluun sekä REST-rajapinnan suunnitteluun. Myös Azuressa toimivien palvelujen toteuttamiseen ja hallinnointiin osallistuvat molemmat tekijät.

Frontend-sovelluksen kokonaisuuteen kuuluu Angular-sovelluksen toteutus. Backend-sovellukseen kuuluvat SQL-tietokannan hallinta, IoT-laitehallintajärjestelmät, sekä .Net Core Web API projekti.

2 Prosessi ja menetelmät

Kappaleessa kuvataan koko ohjelmistokehitysprojektia ympäröinyttä filosofiaa, ketterää kehitystä, sekä millaisia suunnitelmia projektin toteutusta varten tehtiin ketterän kehityksen mukaisesti.

Ketterä ohjelmistokehitys koostuu joukosta menetelmiä, joiden avulla voidaan ohjata ohjelmistotuotantoa. Ketterän ohjelmistokehityksen periaatteena on toimia läheisessä yhteistyössä asiakkaan kanssa ja tuottaa asiakkaille säännöllisin väliajoin toimiva versio sovelluksesta, sekä ottaa mukaan asiakkaalta saatuja toiveita ja tarpeita sovelluksen kehitykseen myös myöhemmissä vaiheissa.

Useimmille menetelmille on tyypillistä, että kehitystyö jaetaan osiin, niin kutsuttuihin sprintteihin, joiden aikana kehitystyötä tehdään. Sprinttien pituus vaihtelee, mutta hyväksi havaittu pituus on yleisesti pari viikkoa kerrallaan riippuen muun muassa kehitystyön laajuudesta, siinä käytettävistä työkaluista ja projektitiimin koosta. Menetelmiä on useita erilaisia mutta tässä keskitytään vain yhteen, Agile-malliin.

Luvussa esitetään, miten menetelmät sitoutuivat koko projektin eri vaiheisiin ja mitä eri työkaluja käytettiin apuna.

2.1 Agile

Ketterien menetelmien historia alkaa 1970-luvulta, kun Winston Royce julkaisi artikkelin, jossa hän esitteli vesiputousmallina tunnetun kehitysmenetelmän, joka perustui hänen kokemuksiinsa. Vesiputousmallissa työvaiheet seuraavat toisiaan: vaatimusten määrittely, analyysi, suunnittelu, ohjelmointi, testaus sekä käyttöönotto. 70- ja 80-luvuilla vesiputousmalli oli pääasiallinen ohjelmistokehitysmenetelmä. 90-luvulla vesiputousmallia alettiin pitää ongelmallisena ja kehitettiin paljon erilaisia kevyempiä menetelmiä. Osassa näistä taustalla oli tuotantoteollisuudesta tullut Lean-ajattelutapa.

Vuonna 2001 Utahissa kirjoitettua ketterän ohjelmistokehityksen julistusta (Agile Manifesto) pidetään usein Ketteryyden alkuperänä. Julistuksen tekivät 17 ohjelmistoalan ammattilaista. Julistus oli kokoelma niistä arvoista, jotka 90-luvulla uusia menetelmiä kehittäneet ohjelmistoammattilaiset olivat kokeneet hyviksi. Julistus koostuu näistä arvoista ja 12 periaatteesta. (Ketterä käsikirja 2015.)

Agile on sateenvarjokäsite joukolle rakenteita, viitekehyksiä ja käytäntöjä, jotka pohjautuvat Agile manifestissa määritettyihin kahteentoista arvoon ja periaatteeseen (Kuva 1). Periaatteet kuvaavat Agilen mukaista ajatustapaa, joiden avulla voi selvittää oikeat toimenpiteet eri tilanteisiin.

Agile keskittyy kehitystä tekeviin ihmisiin ja miten he työskentelevät yhdessä. Johtajien tehtävänä on varmistaa, että tiimillä on, tai he pystyvät hankkimaan tarvittavat taidot. Johtajat ylläpitävät ympäristöä, jossa kehitystiimi voi onnistua. Agile tiimi työskentelee jonkun viitekehyksen puitteissa, josta tiimi muokkaa omanlaisensa työskentelytavan, joka sopii heidän tilanteeseensa. (Agile Alliance 2020.)

Julistuksen takana olevat periaatteet

Noudatamme seuraavia periaatteita:

Tärkein tavoitteemme on tyydyttää asiakas toimittamalla tämän tarpeet täyttäviä versioita ohjelmistosta aikaisessa vaiheessa ja säännöllisesti.

Otamme vastaan muuttuvat vaatimukset myös kehityksen myöhäisessä vaiheessa. Ketterät menetelmät hyödyntävät muutosta asiakkaan kilpailukyvyn edistämiseksi.

Toimitamme versioita toimivasta ohjelmistosta säännöllisesti, parin viikon tai kuukauden välein, ja suosimme lyhyempää aikaväliä.

Liiketoiminnan edustajien ja ohjelmistokehittäjien tulee työskennellä yhdessä päivittäin koko projektin ajan.

Rakennamme projektit motivoituneiden yksilöiden ympärille. Annamme heille puitteet ja tuen, jonka he tarvitsevat ja luotamme siihen, että he saavat työn tehtyä.

Tehokkain ja toimivin tapa tiedon välittämiseksi kehitystiimille ja tiimin jäsenten kesken on kasvokkain käytävä keskustelu.

Toimiva ohjelmisto on edistymisen ensisijainen mittari.

Ketterät menetelmät kannustavat kestäväään toimintatapaan. Hankkeen omistajien, kehittäjien ja ohjelmiston käyttäjien tulisi pystyä ylläpitämään työtahtinsa hamaan tulevaisuuteen.

Teknisen laadun ja ohjelmiston hyvän rakenteen jatkuva huomiointi edesauttaa ketteryyttä.

Yksinkertaisuus - tekemättä jätettävän työn maksimointi - on oleellista.

Parhaat arkkitehtuurit, vaatimukset ja suunnitelmat syntyvät itseorganisoiduvissa tiimeissä.

Tiimi tarkastelee säännöllisesti, kuinka parantaa tehokkuuttaan, ja mukauttaa toimintaansa sen mukaisesti.

2.2 Järjestelmän vaatimuksien määrittely

Kappaleen tavoitteena on selventää, miten vaatimusmäärittelyä voidaan tehdä, mitä hyötyjä vaatimusmäärittelyn tekemisestä on ja mitä työkalua projektissa käytettiin vaatimusmäärittelyn apuna.

2.2.1 Vaatimusmäärittely

Vaatimusmäärittely tehdään yhdessä asiakkaan kanssa ja siinä pyritään vastaamaan kysymyksiin, mitä halutaan ja miksi halutaan. Vaatimusmäärittelyllä pyritään varmistamaan, että valmis sovellus vastaa siltä vaadittuja ominaisuuksia. Vaatimusmäärittelyä voidaan tehdä projektin alussa ja sitä voidaan tarkentaa projektin kuluessa.

Vaatimusmäärittely on dokumentti, jossa kuvataan projektin tavoitteita ja vaatimuksia. Vaatimusmäärittelyssä toteutettavat asiat priorisoidaan tärkeysjärjestykseen, jotta vähintään minimivaatimukset sovelluksen toimivuuden kannalta täyttyvät. Vaatimusmäärittelyssä määritellään kuinka valmiin sovelluksen tulisi toimia ja miten nämä vaatimukset toteutetaan. Sovelluksen toiminnallisuuksia kuvataan käyttötapausten avulla ja ne jaetaan toiminnallisiin, sekä ei-toiminnallisiin vaatimuksiin. Toiminnalliset vaatimukset sisältävät kaiken sen, miten valmiin sovelluksen tulee toimia ja ei-toiminnalliset vaatimukset puolestaan sisältävät kaiken sen, miten sovellus tekee kaiken sen, mitä sen halutaan tekevän. Vaatimusmäärittelyssä tuodaan esille myös projektin aikataulua ja kuinka paljon asiakas on valmis sijoittamaan rahaa projektiin.

2.2.2 Confluence

Confluence on Atlassianin työkalu projektinhallintaan. Confluencen avulla ohjelmointitiimi voi kirjoittaa ylös projektin vaatimuksia, osoittaa tehtäviä tiimin jäsenille ja hallinnoida projektiin liittyviä kalentereita ja aikatauluja samanaikaisesti. (Atlassian 2020a.)

Confluence sivu jakautuu tiloihin, jotka toimivat koontipaikkoina tilaan liittyville dokumenttisivuille. Esimerkiksi tila on sovelluskehitys, johon kerätään sovelluskehitykseen liittyvät dokumentit, kuten vaatimusmäärittelyt ja kokouksien muistiinpanot. (Atlassian 2020b.)

2.3 Käyttöliittymäsuunnittelu

Käyttöliittymäsuunnittelu kehittyy jatkuvasti ja erilaiset trendit ohjaavat suunnittelua. Kehitystä ja trendejä luovat mm. uudet laitteet ja käyttäjien muuttuvat tavat. Siksi käyttöliittymäsuunnittelussa käytetään useasti käyttäjäkeskeistä suunnittelua (User Centered Design),

jossa otetaan huomioon asiakkaan toiveet ja tarpeet koko sovelluksen suunnittelun kehityskaaren aikana.

Uusien laitteiden ja käyttäjien muuttuvat tavat aiheuttavat käyttöliittymän suunnittelussa huomioon otettavia asioita, kuten käyttöliittymän responsiivisuuden suunnittelun (Responsive Web Design) eri laitteille menettämättä kuitenkaan käytettävyyttä. Käyttöliittymän on skaalauduttava puhelimen, tabletin ja tietokoneen ruuduille, jotta tuote on saavutettavissa paikasta riippumatta. (Interaction Design 2020.)

2.3.1 UI/UX suunnittelu

UI-suunnittelu (User Interface Design) eli käyttöliittymäsuunnittelu tarkoittaa sitä, miltä tuote, palvelu tai verkkosivu näyttää loppukäyttäjän näytöllä ja UX-suunnittelu (User Experience Design) eli käyttökokemussuunnittelu tarkoittaa sitä, miltä palvelun käyttäminen tuntuu ja mitä tunteita se herättää. UI ja UX kulkevat käsikädessä, sillä UI-suunnittelijat ovat riippuvaisia UX-suunnittelijoiden päätöksistä ja niiden vaikutuksesta käyttöliittymän ulkoasuun. UX-suunnittelijat puolestaan ovat riippuvaisia UI-suunnittelijoiden ulkoasuun liittyvistä päätöksistä ja niiden vaikutuksesta käyttökokemuksen suunnittelussa.

UI-suunnittelija suunnittelee jokaisen käyttöliittymässä käytetyn elementin ulkoasun, asetelun, siirtymän ja animaation, jotta käyttöliittymä näyttää ja tuntuu hyvältä. Jokaisen elementin on oltava yhtenäinen ja tarkoituksenmukainen muiden elementtien kanssa.

UX-suunnittelija suunnittelee jokaisen elementin paikan ja siirtymät, jotta sovelluksen käytettävyys on jouheva ja tarkoituksenmukainen. Elementtien asettelu oikeisiin kohtiin helpottaa loppukäyttäjän navigointia sovelluksessa ja asettaa sovelluksen toiminnallisuudet loogiseen järjestykseen, joka vaikuttaa helppokäyttöisyyteen.

2.3.2 Suunnittelutyökalu Adobe XD

Adobe XD on vektoripohjainen työkalu käyttöliittymien ja käyttökokemusten suunnitteluun. Työkalu mahdollistaa suunnittelun ja prototyyppien toteutuksen verkkosivustoille ja mobiilisovelluksille. Työkalu mahdollistaa myös suunnittelijoiden samanaikaisen työskentelyn käyttöliittymäsuunnittelun parissa. Prototyyppejä on mahdollista jakaa linkkeinä asiakkaille, jolloin asiakkaat näkevät sovelluksen visuaalisen ilmeen ja toiminnallisen periaatteen käytännössä ilman yhtäkään riviä koodia.

2.4 API-rajapinta ja sen suunnittelu

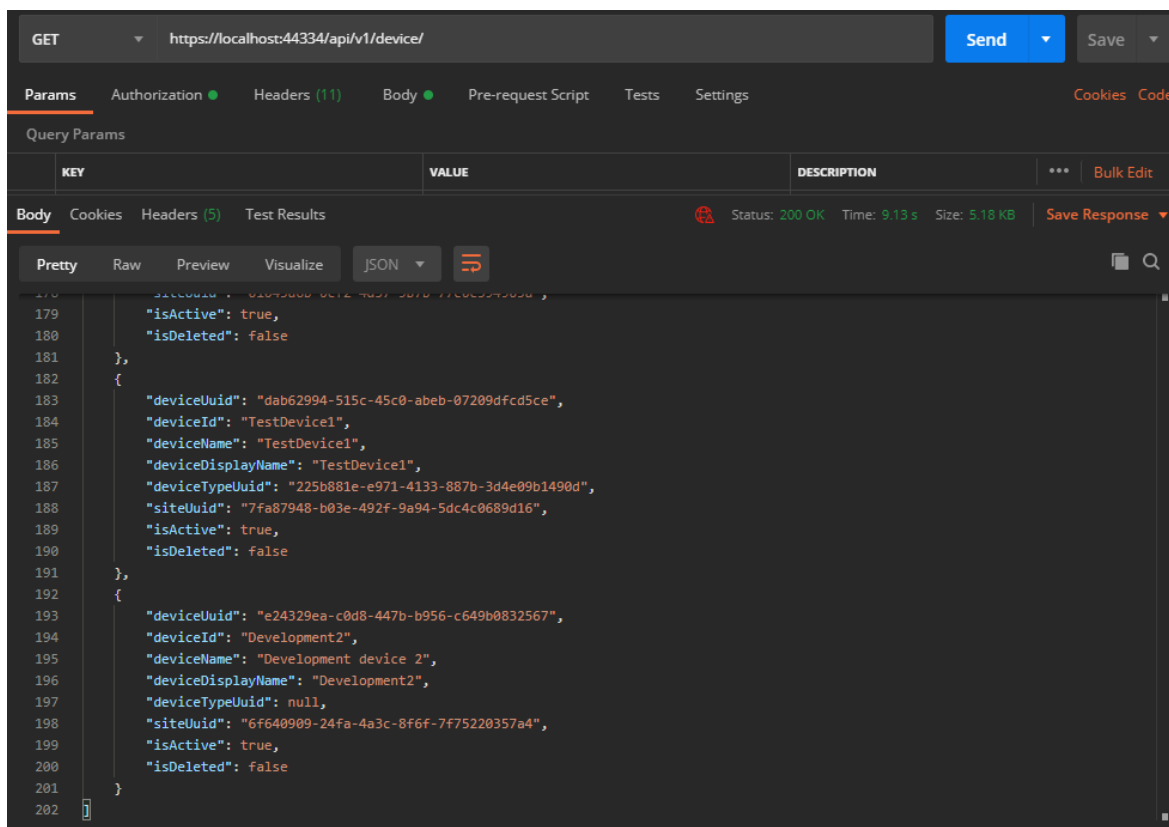
API eli ohjelmointirajapinta, koostuu joukosta määritelmiä, protokollia ja työkaluja, joiden avulla ohjelmistoja voidaan rakentaa tai integroida jo olemassa oleviin järjestelmiin. Ohjelmointirajapintojen avulla ohjelmistot voivat olla yhteydessä muihin ohjelmistoihin, ilman tietoa siitä, millä ohjelmointikielillä ja teknologioilla rajapinnat on kehitetty. Tämän vuoksi rajapinnat yksinkertaistavat ja nopeuttavat sovelluskehitystä, mikä voi säästää ohjelmistoyritysten aikaa ja siten myös kuluja. Ohjelmointirajapintojen yleisimpiä käyttötarkoituksia ovat käyttöjärjestelmät, ohjelmointikirjastot ja Web. (Red Hat 2020.)

2.4.1 REST-rajapinta ja suunnittelun hyviä käytäntöjä

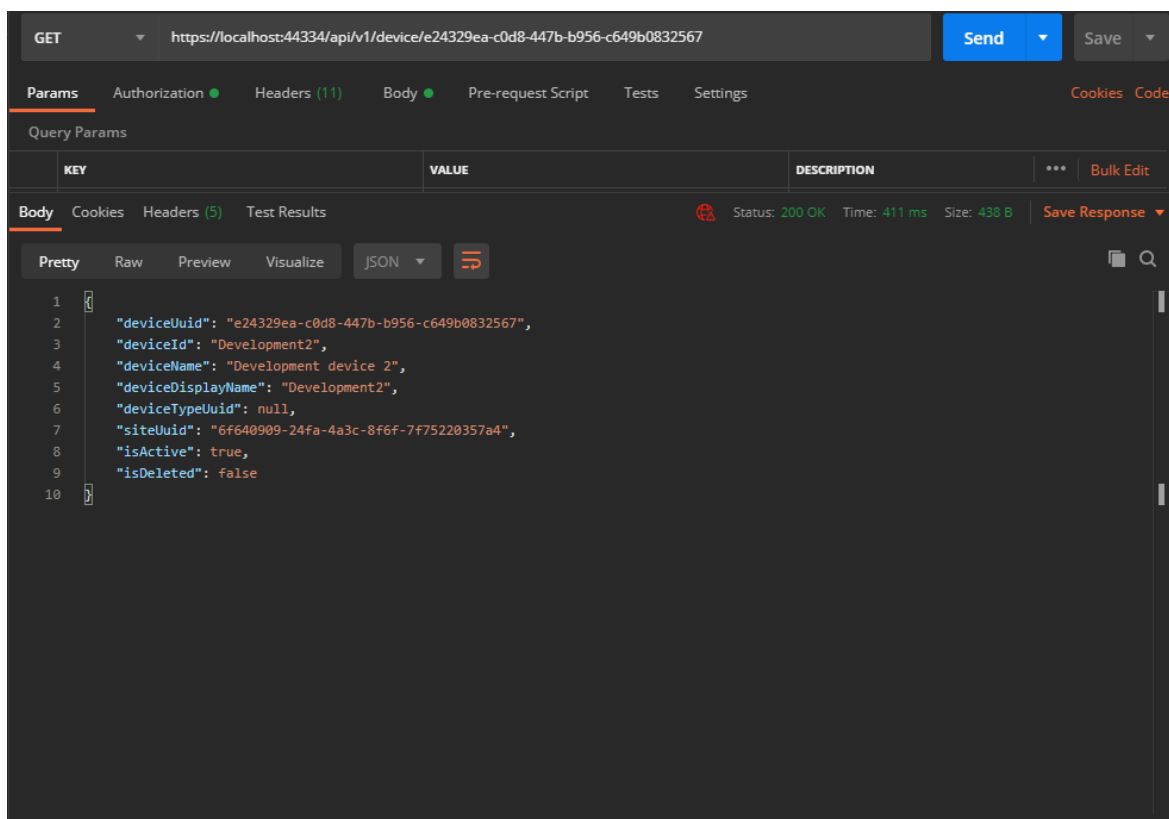
REST (Representational State Transfer) on ohjelmointirajapintojen toteuttamiseen käytetty arkkitehtuurimalli, joka tarjoaa yhteiset standardit sovellusten ja internetin välille, jotta eri systeemien on helpompi kommunikoida keskenään. REST-pohjaisia systeemejä yhdistää se, että ne ovat tilattomia ja erottavat asiakassovelluksen ja serverin (backend-sovelluksen) ongelmat. Ongelmien eriytyminen mahdollistaa sen, että asiakassovelluksen ja backendin toteutukset voidaan tehdä itsenäisesti niiden tietämättä toisistaan, eli koodimuutokset toisessa eivät vaikuta toiseen. REST-rajapinta määrittelee, minkä muotoisia viestejä sovelluksien eri toteutukset voivat lähettää välillään. Tilattomuus REST-rajapinnassa viittaa siihen, että backend-sovelluksen ei tarvitse tietää, missä tilassa asiakassovellus on ja päinvastoin. (Codecademy 2020a.)

Rajapinnan suunnittelun hyviä käytäntöjä voi katsoa monesta näkökulmasta. Ensimmäisenä rajapinnan suunnittelussa on hyvä identifioida rajapintaa kuvaavaan sovellukseen liittyvät objektit. Nämä objektit kuvataan rajapinnassa resursseina. (Restfulapi.net 2020.) Esimerkiksi resurssi voi olla sovelluksen malliluokka Device, joka kuvaa laitetta.

Rajapinnan on hyvä lähettää ja vastaanottaa dataa yleisesti käytetyssä JSON muodossa. Monissa käyttöliittymän ja palvelinpuolen toteutukseen käytetyissä kielissä on valmiit kirjastot JSON datan käsittelyyn, sillä JSON on muodostunut standardiksi datan siirrossa. Reitien määrittämisessä on hyvä käyttää substantiiveja verbien sijasta, pyynnössä määritetty HTTP-metodi määrittää metodin toiminnan. (The Overflow 2020.) Esimerkiksi HTTP GET-pyyntö reittiin /device (Kuva 2) palauttaa kaikki laitteet ja HTTP GET-pyyntö reittiin /device/{id} palauttaa yhden tietyn laitteen (Kuva 3).



Kuva 2. GET HTTP-pyyntö reittiin device, joka palauttaa kaikki laitteet



Kuva 3. GET HTTP-pyyntö reittiin device/{id}, joka palauttaa tietyn laitteen

2.4.2 OpenAPI 3.0

OpenApi spesifikaatio on kuvausformaatti REST-rajapinnoille. Sillä pystyy kuvailemaan rajapinnan päätepisteet eli endpointit, operaatioissa lähtevät ja palautuvat parametrit, autentikointimetodit, sekä sovellukseen liittyviä muita tietoja kuten lisenssin ja käyttöehdot. API spesifikaatioita voi kirjoittaa joko YAML- tai JSON-kielillä. (SmartBear Software 2020.)

OpenApi spesifikaatio määrittelee standardin kielen kuvaamaan REST-rajapintaa, jonka avulla sekä ihmiset että tietokoneet voivat ymmärtää rajapinnan kyvyt ilman tietoa rajapinnan lähdekoodista, dokumentaatiosta tai tarkastelematta rajapinnan yli kulkevien tietoliikennepakettien sisältöjä (OAI 2020).

2.4.3 Postman

Projektissa Postmania käytettiin sekä API-kutsujen tekemiseen, että API-rajapinnan kuvaamiseen API Builder työkalulla.

Postman API Builder on Postmanin tarjoama työkalu, josta löytyy monia toimintoja muun muassa API-rajapinnan kuvaamiseen, versionhallintaan, testaamiseen ja versioimiseen. API-rajapinnan pystyy suunnittelemaan OpenApi 3.0 spesifikaation mukaisesti. (Postman 2020a.) API Builderia pystyy käyttämään sekä Postmanin verkkopalvelusta että Postman sovelluksesta.

Postmanin sovelluksella voi myös tehdä kutsuja API-rajapintoihin, minkä vuoksi sitä voi helposti käyttää sovelluskehityksessä rajapinnan testaamiseen ja vikojen korjaamiseen. Pyyntöillä voi tehdä kaikki samat toimenpiteet, mitä käyttöliittymäsovelluksella pystyisi, eli tiedojen hakeminen, lisääminen, muokkaaminen ja poistaminen. Pyyntöihin pystyy määrittämään ja lähettämään kaikki tarvittavat parametrit, datat ja lupatiedot, joita rajapinta vaatii. Sovellus näyttää rajapinnan palauttaman vastauksen helposti tarkasteltavassa muodossa. (Postman 2020b.)

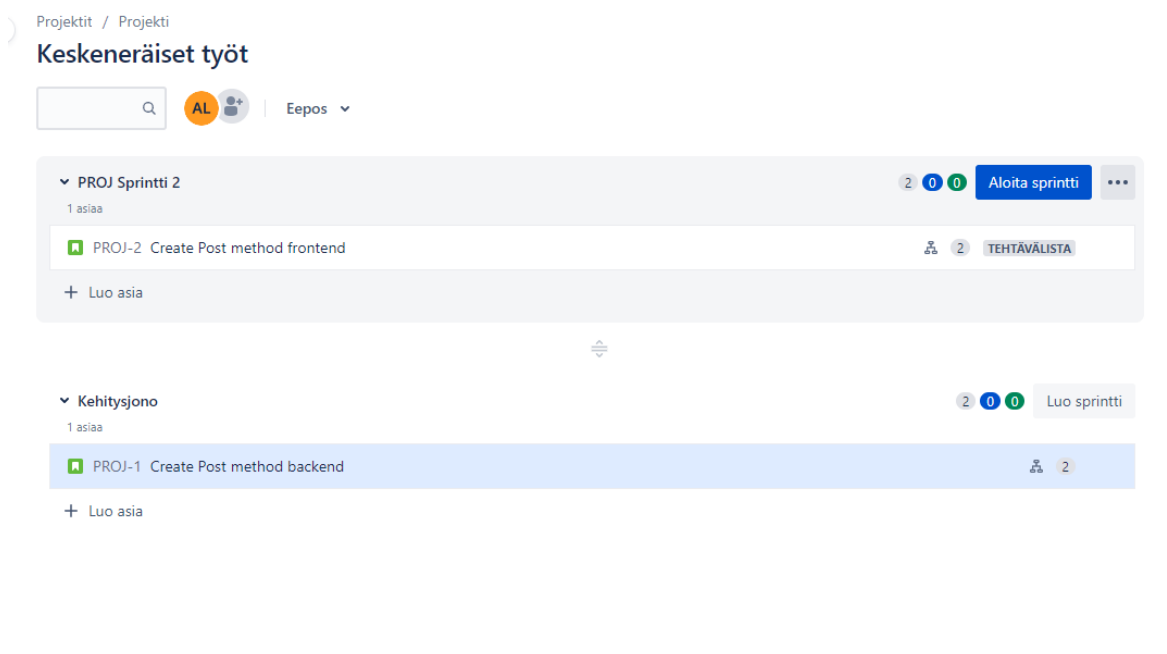
2.5 Projektinhallinta

Projektinhallinta on tärkeässä roolissa ohjelmistokehityksessä, sillä sen avulla määritellään mm. projektin aikataulu, budjetti ja työnjako. Projektinhallinnan avulla ohjelmistokehitys saadaan pidettyä hallinnassa läpi koko projektin ja sillä voidaan minimoida kaikki ohjelmistokehitykseen liittyvät riskit. Ketteriä ohjelmistokehitysmenetelmiä on useita, tässä työssä niistä esitetään projektissa käytetty Scrum-viitekehys.

2.5.1 Scrum

Scrum on yksi ketterän ohjelmistokehityksen mukainen viitekehys, joka auttaa tiimiä yhteistyössä. Scrum painottaa yhdessä oppimista tiiminä, itseorganisoitumista ongelmien ratkaisussa sekä tulosten reflektointia tiimin jatkuvaan parantamiseen. Scrum mallia käytetään usein ohjelmointitiimeissä, mutta sen periaatteet sopivat kaikenlaiseen tiimityöhön. (Atlasian 2020c.)

Tärkeä elementti Scrumissa on tuotteen kehitysjono (backlog), joka jakautuu kahteen osaan. Toinen osa on isompi product backlog, joka kattaa koko sovelluksen tehtävät. Pienempi sprint backlog toimii tehtävienhallinnassa sprinttien aikana. Sprinttien alkaessa product backlogista siirretään tehtäviä sprint backlogiin suoritettaviksi (Kuva 4).



Kuva 4. Product backlog alempana ja ylhäällä sprintin backlog

Backlog kuvastaa priorisoitua listaa kehitystiimin työtehtävistä, jotka ovat johdettu suunnittelusta ja vaatimusmäärittelystä. Backlogiin työ järjestetään Kanbanin jatkuvan kehityksen mukaisesti, tai Scrumin mukaisiin sprintteihin. Backlogin jatkuva tarkastelu ja päivitys ovat tärkeitä tehtäviä, jonka merkitys korostuu entisestään backlogin kasvaessa. Tällöin tehtävien priorisointi on tärkeää. Backlogin hallintaan vaikuttavat Scrum-viitekehityksen roolit. (Atlasian 2020d.)

Scrumissa on kolme roolia, jotka kuvaavat Scrum tiimin eri jäsenten vastualueet. Roolien tarkoituksena on rajata selkeästi tiiminjäsenten vastuut ja auttaa tiimiä organisoitumaan ja

parantamaan työskentelyään. Roolit ovat product owner, Scrum master, sekä kehitystiimin jäsenet.

Product owner eli tuoteomistaja toimii tiimin vetäjänä ja edustaa asiakasta. Päätehtävä product ownerilla on antaa selkeä kuva työn toteuttajille, ja priorisoida mitkä asiat ovat tärkeimpiä ja tuottavat eniten arvoa. Product ownerilla on kolme vastuualuetta projektissa:

1. Product owner vastaa backlogista, jonka vuoksi hänen tulee olla koko ajan olla perillä, mitä tehtäviä backlogilla on.
2. Product owner vastaa sovelluksen julkaisuaikataulusta, koska Scrumin periaatteiden mukaisesti sovellusta julkaistaan useasti. Asiakkaalta saadun palautteen perusteella tuotetta voidaan muuttaa joustavasti.
3. Product owner vastaa sidosryhmien tarpeiden hallinnoimisesta ja ryhmien välisestä kommunikoinnista. Sidosryhmiin kuuluu asiakkaan ja kehittäjien lisäksi myös muita ryhmiä, kuten organisaation johto, sekä tuotteen loppukäyttäjät. Product owner toimii esimerkiksi asiakkaan ja kehittäjien välissä, välittäen asiakkaan tarpeita kehitystiimille ymmärrettävässä muodossa.

Scrum master vastaa Scrumin toteutuksesta, auttaa product owneria määrittämään eniten arvoa tuottavat tehtävät ja auttaa kehittäjiä tuottamaan arvoa, sekä koko Scrum tiimiä kehittämään yhdessä toimimisessa. Scrum master vastaa päivittäisestä työstä, eli daily scrumista. Hän toimii käytännössä product ownerin ja kehitystiimin välissä auttaen molempia osapuolia, sekä kommunikoi osapuolten välisiä tarpeita toisilleen. (Atlassian 2020e.)

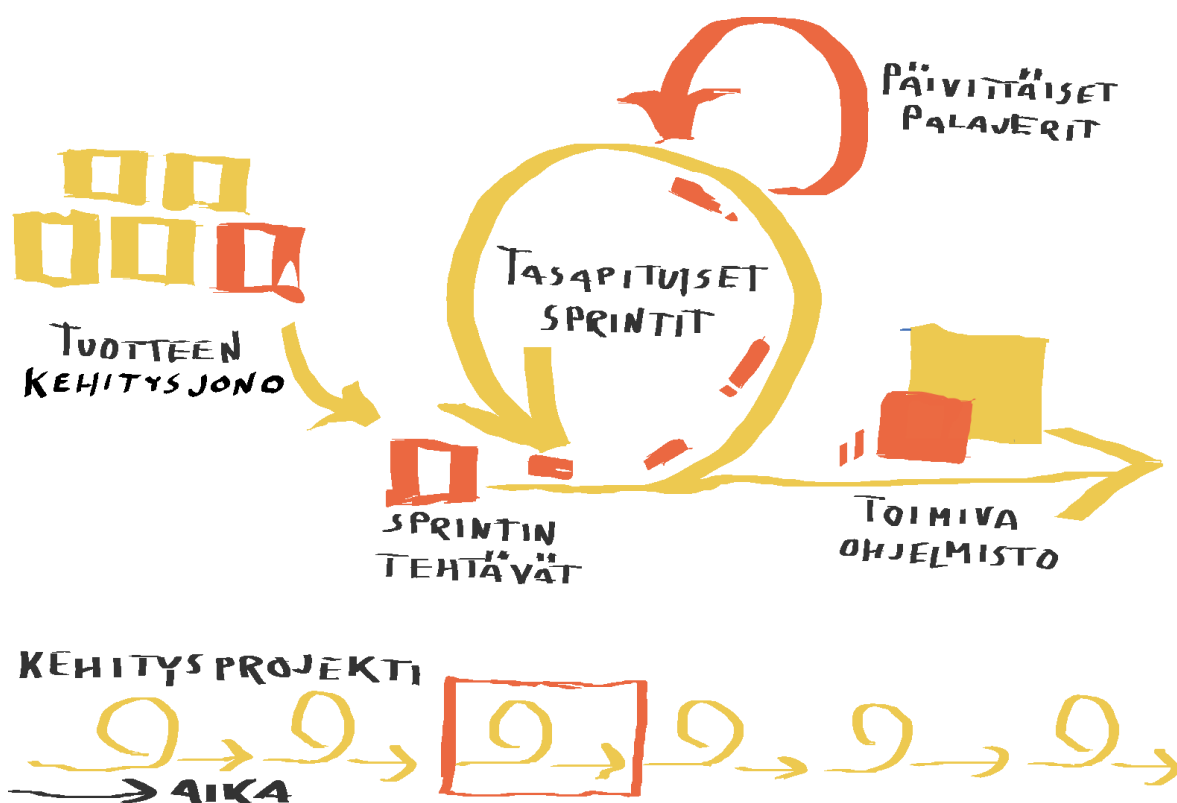
Scrumissa työskentely tapahtuu sprinteissä, jotka ovat lyhyitä projektin laajuuteen sidottuja jaksoja. Jaksojen aikana yritetään saada sprinttiin mukaan otetut asiat toteutettua. Sprintin tarkoitus on toteuttaa laajoja kokonaisuuksia pienemmissä osissa, jotta projekti pysyy helpommin hallinnassa. (Atlassian 2020f.)

Scrum alkaa sprintin suunnittelulla, jossa määritellään, mitä sprintin aikana ehditään tehdä ja kuinka tavoitteet saavutetaan, sekä siirretään tehtävät sprint backlogille. Koko kehitystiimi osallistuu sprintin suunnitteluun ja miettii yhdessä, kuinka sprintti tulisi rakentaa, jotta vaaditut asiat saadaan sprintin aikana toteutettua. Asiat tulisi olla pilkottuna niin pieniin osiin, että ohjelmoija ehtii päivän aikana tekemään vähintään yhden tehtävän sprinttiin asetetuista vaatimuksista. (Atlassian 2020g.)

Sprintin aikana kehitystiimistä jokainen ottaa tehtäviä hoidettavakseen ja näkee samalla projektin edistymisen. Jos projekti ei etene riittävällä nopeudella, voidaan sprintin aikana

pitää kokouksia ja keskustella, kuinka sprintti saadaan toteutettua tarvittavaan aikarajaan mennessä.

Sprintin päättymisen jälkeen on katsaus, jossa näytetään sprintin aikana valmiiksi saatu kokonaisuus. Katsauksessa voidaan käydä läpi ehdotuksia siitä, mitä parannuksia toteutus vielä vaatii ja arvioidaan sprintin kulkua kokonaisuutena, mikä meni hyvin ja mikä huonosti, ja miksi. Kuvassa 5 on esitelty Scrumin mukainen kehitysprojekti.



Kuva 5. Scrum kehitysprojekti (Tech 2015)

2.5.2 Jira

Jira on Atlassianin tehtävienhallintaohjelmisto, joka on tarkoitettu ohjelmistokehitykseen. Jiran avulla isompi projektikokonaisuus voidaan jakaa osatehtäviksi, joille pystytään määrittämään tekijät. Tehtävienhallinnalla huolehditaan, että pienemmät tehtävät tulevat suoritetuiksi, ja näin kokonaisuus valmistuu.

Jirassa projektinhallintaan voi valita erilaisia pohjia, kuten Kanban taulun tai Scrum-viitekehityksen, jossa työ toteutetaan iteroiden sprinteissä. Valinta riippuu usein projektin luonteesta ja tarkoituksenmukaisuudesta.

Jirassa voidaan käyttää tarinapisteitä (story point) hahmottamaan tehtävien laajuutta ja niiden ajallista merkitystä sprintin muihin tehtäviin verrattuna. Mitä suurempi tarinapiste on,

sitä enemmän ohjelmistokehittäjä arvioi, että tehtävään kuluu aikaa toteuttaa tehtävä ja toisinpäin. Tarinapistheet auttavat hahmottamaan myös tulevien sprinttien ajan hallintaa eli sitä, kuinka paljon sprinttiin on mahdollista ottaa mukaan tehtäviä, jotta tehtävät on mahdollista suorittaa ennen sprintin päättymistä. (Atlassian 2020h.)

Yleisesti käytetty menetelmä tarinapisteidien määrittämisessä on Fibonaccin sekvenssi hie-
man ohjelmistokehitykseen muunneltuna riippuen projektin omistajan vaatimuksista. Kes-
kimäärin kahden viikon sprintin aikana olisi hyvä saada suoritettua noin 80 tarinapistettä,
joka vastaa suoraan verrannollisesti ohjelmistokehittäjän työaikaa eli 80 tuntia kahden vii-
kon aikana. Yksi tunti ohjelmistokehittäjän työaikaa on siis suunnilleen yhden tarinapisteen
arvoinen. Fibonaccin sekvenssi on yleisesti muunneltuna muotoon 0, 0.5, 1, 2, 3, 5, 8, 13,
20, 40, 100. Ohjelmistokehittäjät ja kaikki toteutukseen liittyvät henkilöt keskustelevat jokai-
sesta tehtävästä ja niille annetuista pistemääristä, sekä pyrkivät perustelemaan miksi jokin
pistemäärä sopii juuri tälle tehtävälle.

Huomioon on otettava pistemääriä suunniteltaessa se, että vaikka osa tehtävistä on vas-
takkain täysin erilaisia, mutta niihin kuluva aika on suunnilleen sama, tulisi niiden piste-
määrä olla sama, jotta pisteytyksestä saadaan seuraaviin sprintteihin hyötyä. Pistemääriä
arvioitaessa on otettava huomioon myös se, että jos jonkin tehtävän kohdalla pistemäärä
kasvaa yllättävän suureksi, on tehtävää järkevä pohtia uudelleen ja mitoittaa se pienempiin
osiin niin, että tehtävään kuluva aika on realistinen.

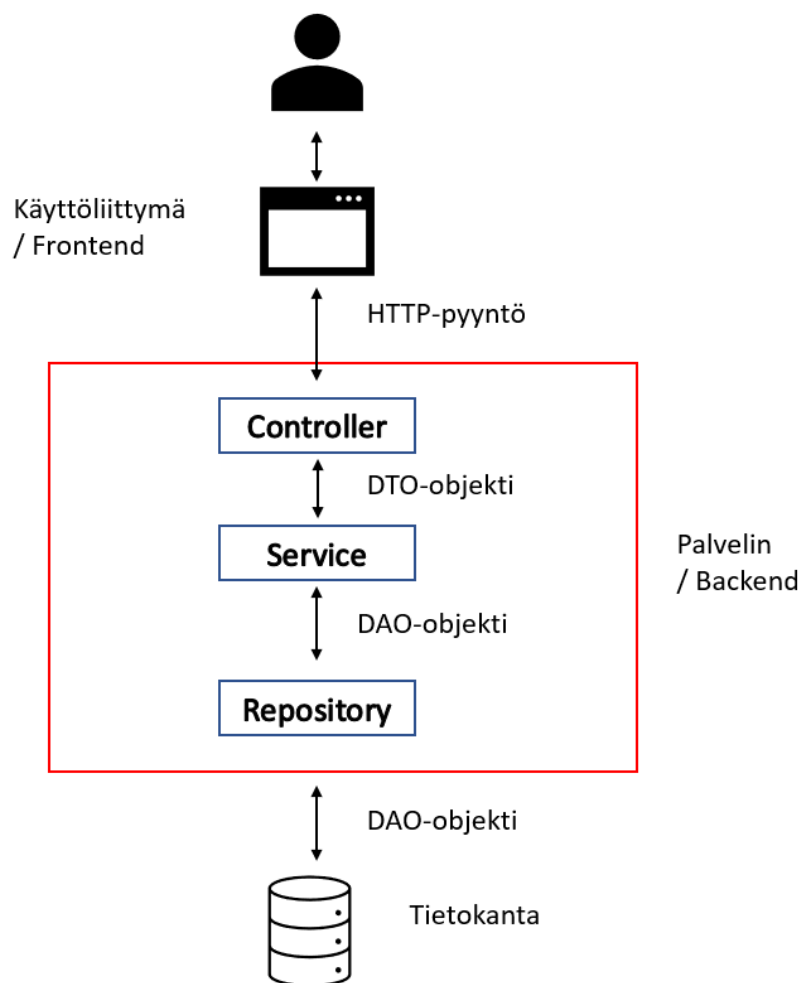
Jirassa päätyneen sprintin jälkeen saadaan erilaisia raportteja sprintin aikana suoritetuista
tehtävistä ja suorittamatta jääneistä tehtävistä. Raporteista on mahdollista selvittää, miksi
sprintti eteni paremmin kuin odotettiin tai päinvastoin. Raporteista on mahdollista nähdä
missä kohtaa sprintti on edennyt aikataulun mukaan tai missä kohtaa aikataulusta on jääty
jälkeen. Useasti jokin odottamaton ongelma aiheuttaa sprintin aikana tulleita ja raporteissa
näkyviä hidasteita. Tällaisia ongelmia voivat olla esimerkiksi jonkin tehtävän uudelleen
suunnitteleminen, koska aikaisemmin suunniteltu tehtävä ei toiminutkaan odotetulla tavalla.

Raporteista on mahdollista nähdä kokonaisvaltainen aikataulu koko projektin osalta riip-
puen siitä, jaetaanko kaikki projektiin liittyvät asiat tehtäviin ennen projektin aloitusta. Jos
asiat jaetaan tehtäviin ja suunnitellaan ennen sprintin alkua, on mahdollista nähdä vain
sprintin aikana toteutunut edistyminen, joka ei ole verrannollinen projektin kokonaiskuvaan.

3 Arkkitehtuurin kuvaus

Kappaleessa esitetään kuvan avulla järjestelmän arkkitehtuurin rakenne ja tutkitaan, miten eri osat liittyvät toisiinsa, millaiset ohjelmointiperiaatteet vaikuttavat rakenteisiin, ja miten järjestelmän eri osat kommunikoivat keskenään.

3.1 Järjestelmän arkkitehtuuri



Kuva 6. Järjestelmän arkkitehtuuri

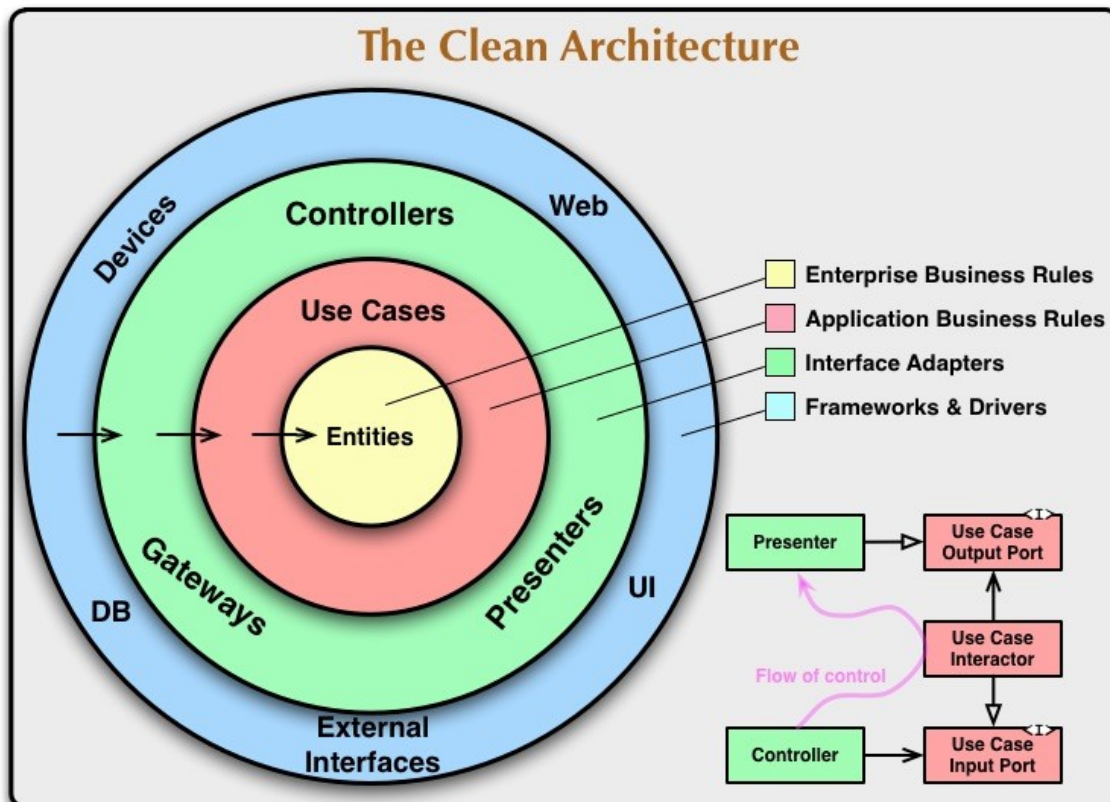
Kuvassa 6 on esitetty järjestelmän arkkitehtuuri. Käyttöliittymäsovellus lähettää HTTP-pyyntöä halutun toimenpiteen palvelinsovelluksen Controller-tasolle. Controllerista pyynnön data liikkuu DTO-objektina (Data Transfer Object) Service-tasolle, ja sieltä muutettuna DAO-objektina (Data Access Object) Repository-tasolle, joka välittää pyynnön tietokantaan suoritettavaksi. Tietokanta palauttaa vastauksen ja edellä kuvattu prosessi tapahtuu käänteisesti käyttöliittymään päin.

3.2 Backend sovelluksen rakenne

Backendin voi rakentaa usealla eri tavalla, mutta yksi suosittu ja yleisesti käytetty arkkitehtuuri on niin kutsuttu Clean architecture. Arkkitehtuurimallin päätavoite on huolenaiheiden erottaminen (separation of concerns), joka saavutetaan jakamalla sovellus kerroksiin ja noudattamalla riippuvuussääntöä. Riippuvuussäännön mukaan arkkitehtuurin ulkoiset osat voivat olla riippuvaisia sisemmistä osista, mutta eivät toisinpäin. Tällä tavoin rakennettu sovellus on helposti testattava ja riippumaton sekä käyttöliittymästä, että tietokannasta. Rajojen ylittävä data saa olla ainoastaan sellaista, joka ei riko riippuvuussääntöä. (The Clean Code Blog 2012.)

Tietokantakyselyt kuitenkin palauttavat niin sanottuja DAO-objekteja, joilla on sisäisiä riippuvuussuhteita, jotka eivät siten sovi Clean malliin. Objektit siis täytyy muuttaa eri muotoon.

Tähän tarkoitukseen voi käyttää yksinkertaisia datarakenteita, kuten DTO-objekteja, joilla ei ole riippuvuuksia tietokantaan tai sovellukseen. Nimensä mukaisesti niitä ei käytetä muihin tarkoituksiin, kuin tiedon siirtämiseen. Muutos DAO-objektista DTO-objektiksi rikkoo riippuvuussuhteet ja auttaa datan siirtelyssä eri tasojen välillä. Muutoslogiikan sijoittamisesta tietylle tasolle on erimielisyyksiä, pitäisikö muutos tehdä Service-tasolla, vai Repository-tasolla. Muutoksen voi tehdä joko manuaalisesti osoittamalla DAO-objektin eri kentät DTO-objektin kenttiin, tai prosessin voi automatisoida erilaisten työkalujen avulla.

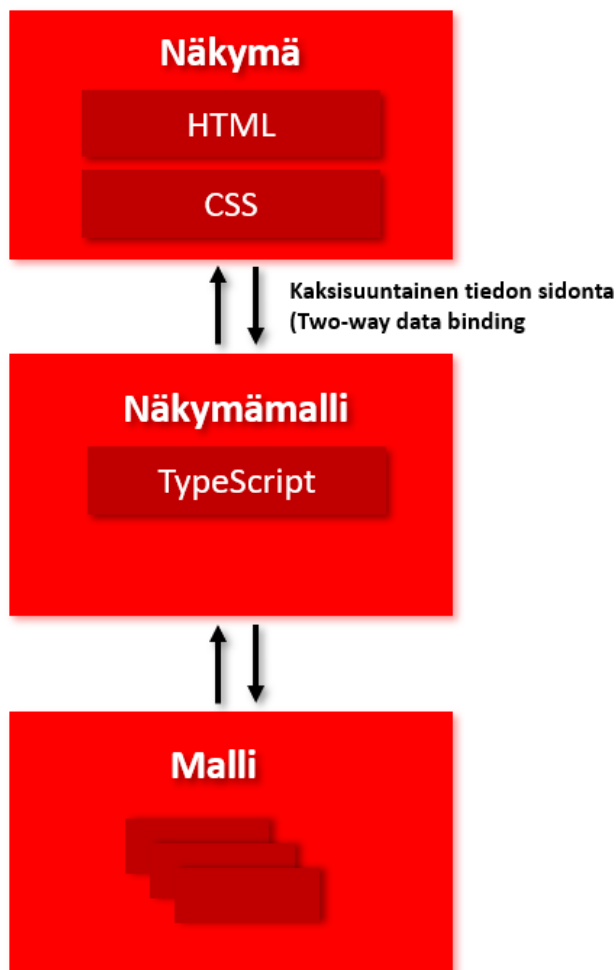


Kuva 7. Clean architecture (The Clean Code Blog 2012)

Kuvassa 7 on esitetty Clean architecturen mukainen sipulimalli. Mallin uloin reuna kuvaa ulkoisia resursseja, jotka ovat yhteydessä palvelinsovellukseen. Näitä ovat muun muassa käyttöliittymäsovellus ja tietokanta. Niiden sijainti ulkoreunalla kuvaa sitä, että niissä tapahtuvat muutokset eivät vaikuta sovelluksen toimintaan. Nuolet kuvaavat riippuvuuden suuntaa tasojen ulkoreunasta sisäänpäin.

3.3 Frontend sovelluksen rakenne

MVVM (Model-View-ViewModel) on yleinen arkkitehtuurimalli Angular SPA (single-page application) kehityksessä. MVVM sopii loistavasti SPA kehitykseen, sillä MVVM:n ansiosta näkymä ja malli voivat olla suoraan yhteydessä toisiinsa näkymämallin avulla, joka helpottaa kommunikointia backendin kanssa. MVVM yhdessä SPA:n kanssa mahdollistaa jatkuvan tiedon tallennuksen tietokantaan. MVVM:n etuihin luetaan myös käyttöliittymän ja sen sovelluslogiikan erottaminen toisistaan, jonka ansiosta ne eivät ole riippuvaisia keskenään. Tämä helpottaa sovelluksen hallitsemista ja kehittämistä. Myös tässä työssä käytettiin edellä mainittua arkkitehtuurimallia. Tähän arkkitehtuurimalliin on yhdistetty REST-rajapinta, jota vasten HTTP-pyyntöjä voidaan lähettää käyttöliittymän puolelta. (Medium 2019.)



Kuva 8. MVVM-arkkitehtuuri

MVVM rakentuu kuvassa 8 esitetystä mallista (Model), näkymästä (View) ja näkymämallista (ViewModel). Malli sisältää sovelluksen datan, jota vasten lähetetään rajapintaan HTTP-pyyntöjä. Näkymä sisältää UI:n eli kaiken sen, mitä sovelluksen käyttäjä näkee käyttöliittymästä. Näkymämalli sisältää frontend-sovelluksen logiikan, jota käyttäjä hallitsee käyttöliittymän kautta. Näkymämallista vaikutetaan malleihin, jotta tietoa saadaan välitettyä rajapinnan kautta backendiin. Malli varmistaa, että haluttu tieto lähetetään oikeassa muodossa eteenpäin. (Hackernoon 2017.)

MVVM:ssä näkymä tietää näkymämallista ja näkymämalli tietää mallista, mutta malli ei ole tietoinen näkymämallista ja näkymämalli ei ole tietoinen näkymästä. Tämän vuoksi näkymämalli eristää näkymän mallista ja antaa mallin kehittyä näkymästä riippumatta, tämä helpottaa myös sovelluksen kehittämistä. (Microsoft 2017d.) Koska näkymämalli ei ole

tietoinen näkymästä, näkymämalli toimii eräänlaisena rajapintana näkymälle, jonka avulla näkymä pääsee käsiksi näkymämallin logiikkaan.

Näkymämalli puolestaan keskustelee palvelun (Service) kanssa, joka määrittää CRUD-operaatiot mallille. Määrittelemällä tehtävät injektoitavaan palveluun, saadaan tehtävät kaikkien komponenttien hyödynnettäväksi. CRUD-operaatiot tulevat sanoista create, read, update ja delete. Create tarkoittaa sovellukseen lisättävää tietoa, read tarkoittaa sovelluksella luettavaa tietoa, update tarkoittaa sovelluksella päivitettävää tietoa ja delete tarkoittaa sovelluksella poistettavaa tietoa. Yleisesti CRUD ilmausta käytetään tietojenkäsittelyssä tietokannan puolella, mutta tässä sillä viitataan palvelun kautta suoritettaviin tehtäviin, jotka määritetään operaatioiden avulla. CRUD-operaatioiden avulla palvelu pystyy määrittämään mallin kautta lähetettävän HTTP-pyynnön rajapintaan. Rajapintaan saadaan yhteys HttpClientin avulla. HttpClient noudattaa HTTP-protokollaa, joka mahdollistaa pyyntöjen lähetyksen tiettyyn rajapinnan osoitteeseen. HttpClient palauttaa observablen, jota voidaan hakea palvelun kautta näkymämalliin ja näkymämallin kautta puolestaan näkymään.

Riippuvuusinjektio (Dependency injection) on yhdistetty Angularin ohjelmistokehykseen ja sitä hyödynnetään, kun komponentilla on tarve päästä käsiksi tarvittavaan palveluun. Riippuvuusinjektio määrittää komponentin konstruktorin parametrityypeissä. Esimerkiksi TestComponent tarvitsee TestServicen injektiona konstruktorin parametreihin, jotta TestComponent pystyy käyttämään TestServicen palveluita ja hakemaan sitä kautta tietoa komponentille.

Tiedon sidonta (Data binding) on keino välittää tietoa näkymiin. Esimerkiksi sitomalla muuttuja käsketään ohjelmistokehyksen (Framework) tarkkailla muuttujaa muutosten varalta. Jos muutoksia havaitaan, ohjelmistokehys huolehtii tiedon päivittämisestä näkymässä.

MVVM:ssä käytetään kaksisuuntaista tiedon sidontaa (Two-way data binding), joka tarkoittaa, että kaikki muutokset joita käyttäjä tekee näkymissä, päivittyy automaattisesti näkymämallin kautta malleihin ja vastaavasti kaikki muutokset, joita malleissa tapahtuu, päivittyy näkymämallin kautta automaattisesti näkymiin. Tieto on siis aina synkronoitu näkymän ja näkymämallin välillä.

Tässä opinnäytetyössä käytetty ohjelmistokehys Angular hyödyntää kaksisuuntaista tiedon sidontaa direktiivillä, jota kutsutaan ngModeliksi. NgModel on osa FormsModuulia ja moduuleita hyödynnetään näkymien rakentamisessa. NgModel mahdollistaa kaksisuuntaisen tiedon sidonnan näkymien ja näkymämallien välillä.

Näkymämalli sisältää sovelluslogiikan, joka koostuu useista funktioista. Funktiot suorittavat tietyn tapahtuman. Kun esimerkiksi käyttöliittymästä painetaan nappia, nappiin on liitetty

funktio, joka suorittaa `subscribe()` -metodin. `Subscribe()` -metodi avaa yhteyden haluttuun palvelun funktioon, joka on tässä tapauksessa `HttpClient.get()` -metodin sisältävä funktio. Palvelun funktio palauttaa HTTP-pyyynnön jälkeen vastauksena observablen. Observable sisältää haetun tiedon, jota voidaan hyödyntää näkymässä kaksisuuntaisen tiedon sidonnan avulla.

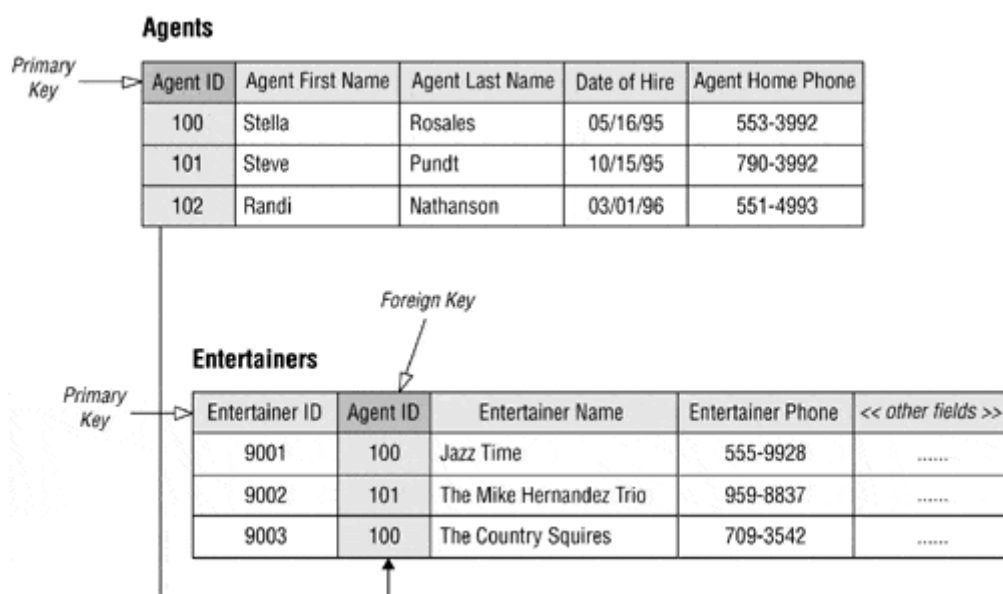
4 Teknologiat

Luvun aiheena on sovelluskehityksessä käytetyt teknologiat. Teknologiat esitellään aloittaen tietokannasta, edeten palvelin puolen toteutukseen ja lopuksi käyttöliittymään.

4.1 Relaatiotietokanta

Relaatiotietokanta on tietyn tyyppinen tietokanta, joka on rakennettu siten, että haluttua dataa voidaan etsiä suhteessa toiseen tietokannassa olevaan dataan. Relaatiotietokannassa data on usein järjestetty tauluihin. Tauluissa data jakautuu riveihin ja kolumneihin. Kolumnit on nimetty kuvaavasti ja niillä on tietyt datatypit, kuten numero tai teksti. Relaatio-tietokantojen hallintajärjestelmät käyttävät usein SQL-kieltä tietokannan hallintaan. (Codecademy 2020b.)

Eri tietokantataulujen tiedot liittyvät toisiinsa primääriavaimella (primary key) ja vierasavaimella (foreign key). Jokaisella taululla tulisi olla yksi primääriavain, joka ei voi olla tyhjä, eikä siitä voi olla duplikaattia. Uutta riviä luotaessa tietokantajärjestelmä antaa uniikin primääriavaimen jokaiselle riville sen tunnistamiseksi (Kuva 9). Vierasavain on toisen taulun kolumni, joka vastaa toisen taulun primääriavainta. Vierasavain määrittää eri taulujen välisen suhteen ja varmistaa että jokaisella vierasavaimella on vastaava kirjaus toisen taulun pääavaimena. (Network Encyclopedia 2020.)



Kuva 9. Primääriavain ja vierasavain (Network Encyclopedia 2020)

4.1.1 SQL Server

SQL Server on Microsoftin tarjoama relaatiotietokannan hallintajärjestelmä, joka suorittaa asiakkaan palvelimen välisen liikenteen SQL-kielellä. Tietokantana sen päätehtävänä on varastoida tietoa ja palauttaa tallennetun tiedon joukosta se, jota asiakasohjelmat haluavat. Asiakasohjelmat voivat sijaita eri paikoissa ja ne voivat tehdä pyyntöjä verkon yli, tai tieto voi sijaita palvelimella itsellään. (Microsoft 2019a.)

4.1.2 SQL Server Management Studio

SQL Server Management Studio on Microsoftin sovellus kaikkien SQL Serverin komponenttien hallintaan. Se toimii graafisena käyttöliittymänä SQL Serverille ja tarjoaa pääsyn SQL Serverin tietoihin ja ohjelmointiin. (Microsoft 2019a.)

4.2 Backend sovellus

4.2.1 .NET Core, C#

.NET on Microsoftin luoma ilmainen vapaan lähdekoodin kehitysalusta, jolla voi rakentaa useita erilaisia sovelluksia, kuten web- ja mobiiliapplikaatioita, pelejä, sekä pilvipalveluita. Sitä ylläpitää Microsoftin lisäksi .NET Foundation yhteisö. .NET on alustariippumaton, mikä merkitsee sitä, että sillä voi luoda ohjelmia muun muassa Windows, Linux, macOS alustoille. Microsoft alkoi kehittää .NET Corea vuonna 2014 ja se toimi tällä nimellä versioon 3.1 asti. Seuraava versio tulee olemaan nimeltään .NET 5. (Microsoft 2020a.)

.NET tukee kolmea ohjelmointikieltä, C#, F# ja Visual Basic. C# on moderni olio-ohjelmointikieli, joka pohjautuu C ohjelmointikieliin. (Microsoft 2020b.)

4.2.2 ASP.NET Core Web Application – Web API

Web API on ASP.NET Core Web Applicationin tarjoama sapluuna Web API sovelluksen luomiseksi. Sovellus käsittelee API-pyyntöjä käyttäen controller-luokkia, jotka pohjautuvat ControllerBase pohjaluokkaan. ControllerBase-luokassa on monia ominaisuuksia ja metodeja HTTP-pyyntöjen käsittelyyn ja controller-luokkien toimintojen konfigurointiin. Ominaisuuksilla voi määrittää esimerkiksi controller-luokan reitin ja halutun HTTP toiminnon. Näitä ominaisuuksia voidaan määrittää lisäämällä erilaisia annotaatioita controller-luokkaan. (Microsoft 2020c.)

Web API-sovellus koostuu yhdestä tai useammasta controller-luokasta, jotka tarjoavat ulospäin näkyvät reitit HTTP-kyselyiden tekoa varten.

4.2.3 NuGet

NuGet on pakettien hallinnointijärjestelmä .NET ympäristölle. Se on välttämätön työkalu nykyaikaisessa ohjelmistokehityksessä ja sen avulla kehittäjät voivat luoda, jakaa ja käyttää hyödyllistä koodia. NuGet paketti on zip-tiedosto, joka sisältää kootun koodin, koodiin liittyviä muita tiedostoja sekä kuvailevan manifestin, josta löytyy muun muassa paketin versio-numero. (Microsoft 2019b.)

NuGet-pakettien avulla ohjelmistokehitys on nopeampaa, sillä kaikkea ei tarvitse ohjelmoida itse, vaan NuGet-kirjastosta löytyy usein valmis työkalu, joka sopii projektin tarpeisiin.

4.2.4 Microsoft Visual Studio kehitysympäristö

Visual Studio integroitu kehitysympäristö on alusta koodin editoimiseen, debuggaamiseen, koodin buildaamiseen ja sovelluksen julkaisuun. Visual Studio eroaa useimmista kehitysympäristöistä siinä, että koodin editoimisen ja debuggaamisen lisäksi siitä löytyy myös muita ominaisuuksia ohjelmointiprosessin helpottamiseksi. Näitä ominaisuuksia ovat muun muassa koodin kääntäjät, työkalut koodin automaattiseen täydennykseen sekä graafisia design työkaluja. (Microsoft 2019c).

4.3 Microsoft Azure

Azure on Microsoftin tarjoama pilvipalvelu. Järjestelmän luomisessa käytettiin monia Azuren palveluja. Käytetyt palvelut esitellään tässä luvussa.

4.3.1 Azure App Services

Azure App Service on HTTP-pohjainen palvelu web-sovellusten, REST-rajapintojen ja backend-sovellusten julkaisuun ja hostaamiseen. App Service tukee monia eri ohjelmointikieliä ja toimivat joko Windows tai Linux-ympäristössä. Ne ovat helposti skaalattavissa sovelluksen koon mukaan. (Microsoft 2020d.)

4.3.2 Azure AD

Azure AD on Microsoftin pilvipalvelu Microsoft tilien ja tilien pääsyoikeuksien hallintaan. Palvelu on tarkoitettu esimerkiksi yritysten IT admineille, jotka voivat hallinnoida työntekijöiden käyttöoikeuksia ja tilejä, sekä ohjelmointikehitykseen sisäänkirjautumisen luomista varten. (Microsoft 2020e.)

4.3.3 Azure Pipelines

Azure Pipelines on Azuren DevOps palvelusta löytyvä CI/CD (jatkuva integraatio, jatkuva julkaisu) pilvipalvelu, joka mahdollistaa sovellusten automaattisen buildauksen, testauksen ja julkaisun. Palvelu tukee monia eri ohjelmointikieliä ja integroituu useimpien eri versionhallintajärjestelmien kanssa. (Microsoft 2019d.)

4.4 IoT-laitehallinta järjestelmät

Projektissa käytettiin kahta IoT-laitehallintaan tarkoitettua järjestelmää. Toinen mahdollistaa viestien välityksen pilvipalveluiden ja laitteiden välillä, ja toinen jakaa ja hallinnoi laitteiden käyttämää ohjelmistoa. Tämän lisäksi järjestelmästä voi hallinnoida laitteisiin liittyviä parametreja, joilla voidaan vaikuttaa muun muassa laitteissa toimivan ohjelmiston ulkonäköön ja muihin ominaisuuksiin.

4.5 Frontend sovellus

4.5.1 Angular Framework

Angular Framework on avoimen lähdekoodin kehitysympäristö SPA (Single Page Application) sovellusten luomiseen HTML:llä ja TypeScriptillä. Angular SPA sovellukset toteutetaan TypeScript-kielellä. Sovelluksen peruselementtejä ovat komponentit, jotka rakentuvat moduuleista. Angularissa on sisäänrakennettuja valmiita moduuleja, joita Angular pyytää asentamaan sovelluksen luomisen yhteydessä. Näitä moduuleja Angular käyttää oletuksena. Sovellukseen on mahdollista asentaa kolmannen osapuolen kirjastoja, jotka ovat saatavilla myös moduuleina.

Komponentit ovat sovelluksen osia, jotka määrittävät sovelluksen käyttäjälle näkyvien sivujen toiminnallisuuden. Komponentit rakentuvat HTML, CSS ja TypeScript-tiedostoista. HTML on verkkosivun rakenne eli se mitä sovelluksen käyttäjät näkevät käyttöliittymässä. CSS on tyylitiedosto, joka määrittää sen, miltä käyttöliittymässä olevat elementit näyttävät. TypeScript on tiedosto, jossa sivun logiikka tapahtuu eli esimerkiksi käyttäjän painaessa hiirellä elementtiä, elementtiin liitetty funktio toteuttaa tietyn käskyn. Elementit rakentuvat moduuleista, jotka sijaitsevat komponenttien HTML-tiedostoissa. (Angular 2020.)

4.5.2 Angular SPA application

Angular SPA (Single Page Application) sovellus on kirjaimellisesti vain yhden sivun sovellus. SPA-sovelluksen toiminta perustuu siihen, että sovelluksen eri näkymiin siirryttäessä ei jokaista sivua ladata kokonaan uudelleen, vaan ainoastaan sivulle haettava sisältö ladataan

sivun vaihdon yhteydessä. Sovelluksen perusnäköymä pysyy siis ennallaan, mutta sovelluksen näköymään liitetty näytettävä sisältö ladataan backendistä HTTP-pyyntöjen avulla näkömaakohtaisesti. (Angular University 2020.)

4.5.3 Node Package Manager

Node.js on avoimen lähdekoodin työkalu sovelluskehitykseen. Nodella voidaan pystyttää lokaali ympäristö eli tietokoneella sijaitseva ohjelmointiympäristö sovellukselle kehitystyön ajaksi. Näin sovellusta voidaan ajaa ympäristössä, joka näyttää ja tuntuu samalta, kuin oikea sovellus, mutta ilman pysyvää osoitetta.

Nodessa on sisäänrakennettu ominaisuus pakettien hallintaan NPM:n (Node Package Manager) avulla. NPM sisältää miljoonia avoimen lähdekoodin paketteja, joita voidaan asentaa kehityksessä olevaan sovellukseen Node CLI:n (Command Line Client) avulla. CLI on Noden sisäänrakennettu ominaisuus. CLI on ohjelmointiympäristössä komentoriville syötettävä käsky, joka mahdollistaa pakettien lataamisen NPM:stä vaivattomasti. (Simform 2020).

4.5.4 JetBrains WebStorm kehitysympäristö

WebStorm on IDE (Integrated Development Environment), joka on suunniteltu lähtökohtaisesti JavaScript ja TypeScript-kielille, mutta se tukee myös useita muita kieliä. WebStorm tukee myös kaikkia suosituimpia kehitysympäristöjä aina Angularista Reactiin. WebStorm on suunniteltu toimimaan yhdessä Noden kanssa, jonka ansiosta se on hyvin suosittu etenkin Angular-ohjelmistokehityksessä. WebStormista löytyy myös kattavat testausominaisuudet yksikkötestaukseen usealla eri työkalulla valitusta kehitysympäristöstä riippuen. (JetBrains WebStorm 2020).

4.6 Versionhallinta

Lähdekoodi on ohjelmistokehityksessä tärkeää omaisuutta ja sitä pitää suojata. Versionhallintajärjestelmät ovat työkaluja, joiden avulla voidaan hallita lähdekoodin muutoksia ajan kuluessa, tallentamalla koodin muutokset erityiseen tietokantaan. Ohjelmistoprojektissa koodia muutetaan koko ajan ja koodi on usein organisoitu eri haaroihin, joissa voi tapahtua samaan aikaan hyvinkin erilaista työtä, kuten eri ominaisuuksien kehittämistä tai jonkin ongelman korjaamista. Tämä koodin muuttaminen eri paikoista samanaikaisesti voi johtaa konflikteihin, kun muutokset eivät ole yhteensopivia keskenään. Versionhallinta seuraa joikaista eri haaraa ja kehittäjää, ja auttaa estämään konflikteja, joita voi tulla samanaikaisesti tehtävistä muutoksista.

Hyvä versionhallintajärjestelmä ei pakota kehittäjiä työskentelemään millään tietyllä tavalla, ja on yhteensopiva riippumatta käytetystä käyttöjärjestelmästä tai ohjelmointikielestä. Versionhallintajärjestelmä on jokaisen modernin ohjelmointitiimin päivittäinen työkalu. (Atlassian 2020i.)

4.6.1 Git

Git on versionhallintatyökalu, joka tallentaa ja säilyttää jokaisen Git-manageroituun tiedostoon tehdyn muutoksen. Git on suunniteltu Linux Kernelille ja sopii ohjelmistokehitykseen. Git tallentaa tiedostot tietovarastoon (repository), joissa voi olla tallennettuna yksi projekti tai monimutkaisempi rakenne. Työnteko repositoryssa jakautuu brancheihin eli koodihaaroihin. Brancheja voidaan jakaa monella tavalla, esimerkiksi jokaisella tiimin jäsenellä voi olla oma branch tai jokaiselle tehtävälle tehdään oma branch. Yleisesti projektissa on kuitenkin yksi master-branch joka toimii projektin pääkoodihaarana. Muutoksille koodissa tehdään commit, eli tallennus, joka tallentaa koodin ja muita tietoja, kuten käyttäjän tiedot ja vaihtoehtoisen viestin, joka kuvaa commitin sisältöä. Commiteille tehdään push, joka lähettää muutokset tietovarastoon. (QuadraNet 2019.)

4.6.2 BitBucket

BitBucket on Atlassianin tarjoama Git-tietovarastojen hallintajärjestelmä, joka tarjoaa keskitetyn paikan hallita Git-tietovarastoja, tehdä koodia yhteistyössä sekä ohjata kehitysprosessia. BitBucket tarjoaa monia hyödyllisiä ominaisuuksia tietovarastojen hallintaan kuten kulunvalvontaa, jolla rajoitetaan pääsyä koodiin, pull requestit joiden avulla voidaan tehdä koodikatselmointia, eli tarkistaa koodin toimivuus ennen koodihaarojen yhdistämistä, ja integraatiota Jiraan, jonka avulla Jiran tehtävät voidaan liittää committeihin. (Atlassian 2018.)

5 Projektin toteutus

Luvussa kerrotaan, miten edellisissä luvuissa esitetyt projektin suunnitteluun ja hallintaan liittyvät menetelmät sekä käytetyt teknologiat tulivat esiin käytännön toteutuksessa.

Alalukujen 5.1–5.3 aiheina ovat suunnitteluun liittyvät vaiheet. Ensimmäisenä pureudutaan projektin alussa tehtyyn vaatimusmäärittelyyn ja siihen liittyviin työkaluihin. Sitten esitellään UI/UX suunnittelun vaiheet, miten suunnittelu tapahtui, miten käyttöliittymänäkymien suunnittelu vaikutti sovellusten toteutukseen ja miten suunnitelmat tarkentuivat työn edetessä. API-suunnittelu luvussa kerrotaan, miten rajapintojen suunnittelu tapahtui, miten työkalu valittiin ja miten se sopi suunnitteluun.

Alaluvussa 5.4 esitetään ketterän ohjelmistokehityksen mukaisen viitekehyksen Scrumin toteutus ja projektinhallinnassa käytettyjen työkalujen käytännöt.

Viimeisen alaluvun aiheena on sovelluskehitys. Luvussa tutkitaan, miten sovelluskehitys tapahtui käytännössä, miten projektinhallinta vaikutti kehitystyöhön sekä miten päivittäinen kehitystyö tapahtui.

5.1 Vaatimusmäärittely

Projektin alussa tehtiin vaatimusmäärittely asiakkaalta saadulle dokumenttipohjalle, jossa oli määritelty kattavasti eri otsikoita monista eri järjestelmäsuunnittelun näkökulmista. Karkeasti jaoteltuna dokumentin sisältö jakautui seuraavasti:

Yleiskuvaus: yleiskuvaus järjestelmän tarkoituksesta, asiakkaasta, järjestelmän toimintaympäristöstä ja käyttäjistä sekä liittymistä muihin järjestelmiin.

Toiminnalliset vaatimukset: yleiskuvaus järjestelmän toiminnasta, käyttäjien ongelmakuvaukset, käyttötapauksien kuvaus, mahdolliset lisätoiminnot ja ei-toteutettavat toiminnot.

Ei-toiminnalliset vaatimukset: käytettävyys, tietoturva, toimintavarmuus, ylläpidettävyys ja huollettavuus sekä siirrettävyys ja laajennettavuus.

Muut vaatimukset: suorituskyky, arkkitehtuurikuvaus, rajapinnat, käyttöliittymät.

Rajoitukset suunnittelulle ja toteutukselle: laitteistorajoitukset ja ohjelmistorajoitukset.

Tiedot ja tietokannat: tallennettavat tiedot.

Alustava aikataulu.

Dokumentin täyttämässä harkittiin, mitkä otsikot olivat järjestelmän suunnittelun kannalta tarkoituksenmukaisia, ja mistä olisi eniten hyötyä järjestelmän määrittelyn kannalta. Suunnitteluun ei haluttu käyttää liikaa aikaa, koska se olisi ollut pois sovelluskehityksestä.

Vaatimusmäärittelydokumenttiin kerättiin toteutettavan järjestelmän kaikki halutut ominaisuudet, jotka määritettiin yhdessä asiakkaan kanssa. Määritellyt ominaisuudet laitettiin tärkeysjärjestykseen sen mukaan, mitkä olivat tärkeimpiä järjestelmän ensimmäisen tuotantoversion kannalta.

Dokumentista sai kokonaiskuvan toteutettavasta järjestelmästä ja sen vaatimuksista. Vaatimusmäärittelydokumentti lisättiin Confluence työskentelyalustalle pohjaksi, johon tuleva työskentelyn tarkempi suunnittelu pohjautui.

5.2 UI/UX suunnittelu

Osana vaatimusmäärittelyä toteutettiin myös käyttöliittymänäkymien suunnittelu. Työkaluksi valittiin Adobe XD, koska se oli ennestään tuttu, ja sen avulla voi helposti yhdistää suunnitteluun myös käyttäjäkokemuksen (UX) puolen.

Käyttöliittymän päänäkömät pyrittiin hahmottamaan ja niistä tehtiin muutama eri versio, jotka erosivat toisistaan värien ja muiden design-elementtien osalta. Versioista valittiin asiakkaan kanssa paras ja näkömät päätyivät jatkokehitykseen, jossa mietittiin sovelluksen käyttöä käyttäjän näkökulmasta, esimerkiksi nappien sijoittelua sekä miten eri näkömiin pääsee ja miten tietyt operaatiot etenevät käyttöliittymässä.

Suunnitellut näkömät yhdistettiin Adobe XD:n prototyypitoiminnon avulla ja niissä olevat napit reititettiin toisiinsa, jonka avulla toteutettavasta käyttöliittymästä saatiin interaktiivinen malli. Kun malli oli valmis ja hyväksytty, aloitettiin sovelluskehitys.

Sovelluskehityksen aikana UI/UX suunnittelu jatkui tarpeen mukaan. Tarpeita tuli esimerkiksi siirryttäessä järjestelmän uusien osa-alueiden toteutukseen, joissa oli osin erilaisia toimintoja perustoiminnallisuuden lisäksi. Uusien toimintojen näkömät mallinnettiin, jotta saatiin nopeasti kuva, millaista käyttöliittymää toiminto vaatii ja miten käyttöliittymä toimii. Monissa tapauksissa suunnitelman mukainen käyttöliittymä vaikutti myös backend-sovelluksen toiminnallisuuteen, esimerkiksi HTTP-pyyynnössä lähetettävien parametrien osalta.

5.3 API suunnittelu

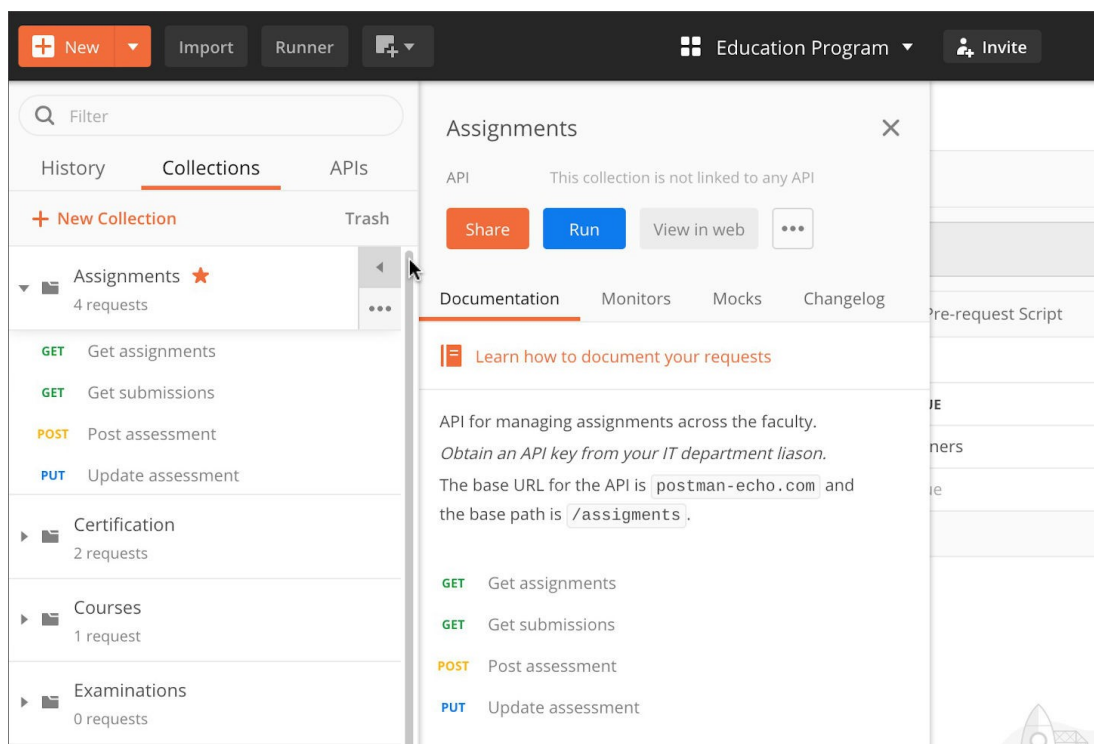
Myös rajapintojen suunnittelu aloitettiin vaatimusmäärittelyn kanssa yhtä aikaa. Suunnittelu tapahtui osissa, aina sovelluskehityksen edetessä uudelle osa-alueelle. API suunnittelu alkoi sopivan työkalun etsimisellä. Pääkriteereinä työkalun valintaan olivat helppokäyttöisyys,

jotta työkalun opettelemiseen ei tarvitse käyttää esimerkiksi viikkoja aikaa, sekä työkalun tunnettuus ja suosio, joka varmistaa osaltaan sen, että ohjelmistoon tulee päivityksiä ja se on ylläpidetty.

Näiden perusteella työkaluksi valikoitui Postmanin tarjoama API Builder, jonka käyttöönotto vaati OAS-spesifikaation opettelua. Työkalun avulla API-rajapinta jaettiin järjestelmän eri osien mukaan omiin kuvauksiinsa, jotka toteutettiin YAML-kielellä. Kuvauksessa esiteltiin muun muassa rajapinnan reitti, mitä rajapinta palauttaa ja mitä sinne lähetetään. Kaikki erilliset API kuvaukset kerättiin myös yhdeksi kootuksi rajapintakuvaukseksi.

Kokemuksen perusteella rajapintojen kuvaus käytetyllä työkalulla oli raskasta ja dokumenteista tuli vaikeasti ylläpidettäviä. Lisäksi niiden lukeminen ei ole nopeaa eikä niistä käy nopeasti selville, miten rajapinta toimii. Tältä osin rajapintojen suunnittelu auttoi lähinnä vain sovelluskehityksessä, mutta ei tuottanut parhaita mahdollisia dokumentteja muuhun käyttöön.

Rajapintojen suunnittelun olisi voinut tehdä myös helpommin Postman työkalun avulla, tekemällä kaikki rajapintaan liittyvät kutsut ja tallentamalla ne sovellukseen kokoelmaksi Collections välilehden alle. Tallennetuilla kutsuilla voi tämän jälkeen tehdä monta toimintoa, kuten julkaista ne, tehdä niiden avulla automaattista testausta tai luoda niistä automaattisesti dokumentaatio API-rajapinnasta (Kuva 10).



Kuva 10. Postman Collections välilehti (Postman 2020c)

5.4 Projektinhallinta

Projektin hallinta jakautui kahteen eri mittakaavaan. Suuremmassa mittakaavassa hallittiin järjestelmän vaatimuksiin ja suunnitteluun liittyviä asioita, ja pienemmässä mittakaavassa päivittäistä ja viikoittaista kehitystyön hallintaa. Molemmat liittyivät toisiinsa ja molemmissa käytettiin omia työkalujaan.

Projektinhallinnan suurempi mittakaava toteutettiin Confluencella, johon kerättiin kaikki suunnitteluun liittyvät materiaalit ja joiden pohjalta järjestelmän tarkempaa toteutusta alettiin suunnitella. Järjestelmä jaettiin Agile määritelmien mukaisesti erillisiin toiminnallisiin kokonaisuuksiin (Epic), jotka jaettiin pienempiin käyttäjätarinoihin (User stories), joista kävi ilmi käyttäjän näkökulmasta haluttu ominaisuus.

Projektissa toiminnallisia kokonaisuuksia oli yhteensä kuusi, joista viisi liittyi järjestelmän eri osiin ja yksi sovelluskehityksen tarpeisiin, kuten ohjelmointiympäristöjen sekä pilvipalveluiden konfigurointiin.

Järjestelmän tulevat käyttäjät jaettiin erilaisiin käyttäjäpersooniin, joista esimerkkinä yrityksen pääkäyttäjä. Persoonien pohjalta käyttäjätarinat kirjoitettiin yksinkertaisiksi lauseiksi, joista käy ilmi persoona, mitä hän haluaa pystyä tekemään järjestelmässä ja miksi.

Esimerkiksi:

Epic: Käyttäjätietojen hallinnointi.

Rooli: Yrityksen pääkäyttäjä.

Käyttäjätarina: Yrityksen pääkäyttäjänä haluan, että voin muokata asiakkaan tietoja tarvittaessa, jotta voin korjata mahdollisia virheitä.

Käyttäjätarinan tietojen avulla määritettiin Jiraan tehtävä, jolla kyseinen käyttäjätarina tulee täytetyksi. Tässä kohdassa projektinhallinnan mittakaava muuttui pienemmäksi ja käytännönläheisemmäksi.

Kehitystyötä lähdettiin tekemään sprinteissä. Sprintin alussa pidettiin sprintin suunnittelukokous (Sprint planning), jota ennen toteutettavana oleva epic oli jaettu toteutettaviksi tehtäviksi (Task).

Tehtävät ovat yleiskuvaus halutusta ominaisuudesta ja ne jakautuvat suoritettaviin, tarkemmin kuvattuihin alatehtäviin (Sub task).

Esimerkiksi:

Tehtävä: Asiakkaan poistaminen.

Alatehtävä: Luo tarvittavat toiminnallisuudet backend-sovellukseen asiakkaan poistamiseksi.

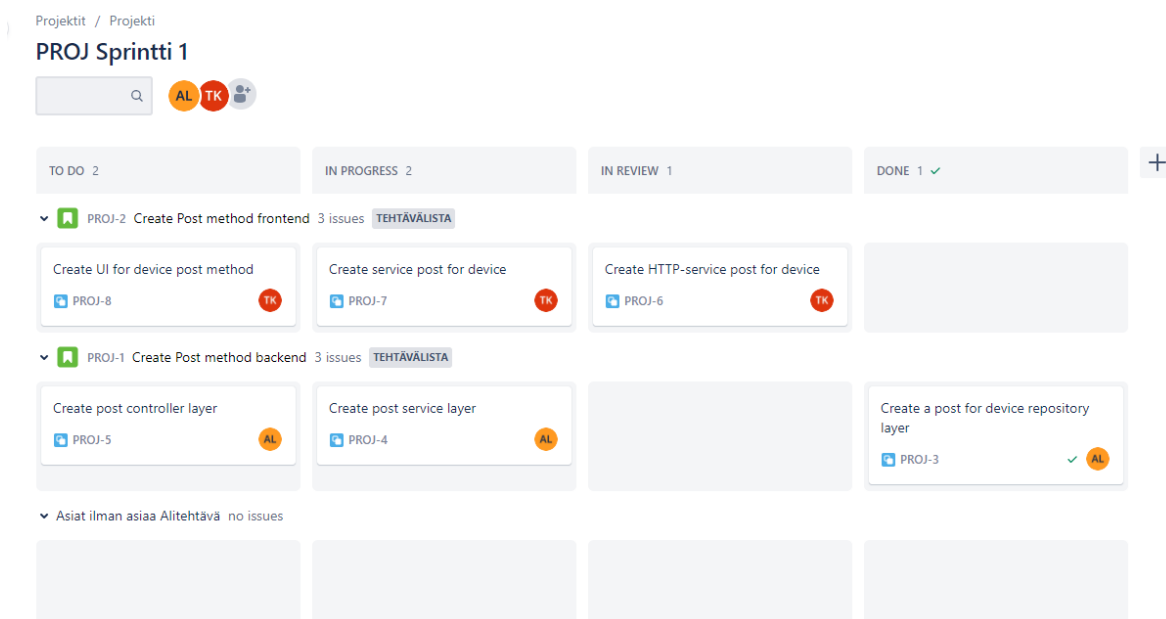
Tarkoituksena tehtävien jaolla on nopeuttaa kehitysprosessia. Parhaimmillaan hyvin kuvattun tehtävän ansiosta kehittäjä voi nopealla vilkaisulla saada tarvittavan yksityiskohtaisen tiedon tehtävän toteutukseen ja näin toteuttaa sen nopeasti.

Suunnittelukokouksessa määritettiin sprintin pituus, kuinka monta erillistä tehtävää sprintin tavoitteena on toteuttaa, sekä erillisten tehtävien pisteytys. Tehtävien pisteytyksen tarkoituksena on määrittää pistemuodossa tiettyyn tehtävään kuluva aika. Tämä opettaa kehitystiimiä arvioimaan omaa ajankäyttöään hahmottamalla, kuinka kauan tietynlaiset tehtävät vievät aikaa. Hahmotuskyvyn karttuminen taas auttaa tulevaisuuden sprinteissä määrittämään, kuinka monta tehtävää sprinttiin kannattaa ottaa.

Tehtävien pisteytys on oiva työkalu myös ammattitaidon kasvattamiseen, koska se auttaa arvioimaan oman osaamisen perusteella, kuinka kauan tuttujen toimintojen ohjelmoimiseen kuluu aikaa ja kuinka kauan uusiin asioihin kannattaa varata aikaa.

Suunnittelukokouksen lopuksi sprint laitettiin käyntiin Jirassa (Kuva 11). Sprinttiä käynnistäessä määritetään sprintin kesto, sekä sprintin tavoite. Kun sprint on aloitettu, toteutettaviksi määritellyt tehtävät tulivat näkyviin tehtävätaululle (backlog). Backlog oli jaettu neljään osaan, toteutettava (To do), toteutuksessa (In progress), arvioitavana (In review) ja valmis (Done) (Kuva 12). Backlogin toiminnasta kerrotaan lisää seuraavassa alaluvussa.

Kuva 11. Sprintin käynnistäminen



Kuva 12. Sprintin tehtävätaulu

Sprintin loputtua pidettiin arviointikokous (Sprint review), jossa todettiin tehtävien pisteytysten avulla, paljonko sprintistä saatiin tehtyä, sekä reflektointia tuloksista eri näkökulmista. Jira tarjoaa monia erilaisia raportteja tulosten vertailuun, joita myös käytettiin hyväksi arvioinnissa.

Arvioinnin jälkeen sprint päätettiin, ja mahdollisesti tekemättä jääneet tehtävät siirtyivät automaattisesti seuraavan sprintin tehtävälistalle. Tämän jälkeen seuraavan sprintin alku-suunnittelu aloitettiin tekemällä lisää tarvittavia tehtäviä, jotka jaettiin alatehtäviin ja tekemällä pisteytyksiä etukäteen. Näin suunnittelu ja toteutus oli jatkuvaa ja sovelluskehitys eteni Scrumin mukaisesti iteraatioissa.

Projektissa sprinttien pituus oli keskimäärin viikon, joka havaittiin toimivaksi pienelle, kahden henkilön kehitystiimille. Viikon työmäärä oli helpompi arvioida eikä tullut tilanteita, joissa työmäärä oli liian yli- tai alimitoitettu. Tämä piti kehityksen tahdin nopeana ja muutoksiin pystyttiin vastaamaan nopeasti.

5.5 Sovelluskehitys

Sovelluskehitys alkoi sprintin käynnistyessä. Jirassa toteutukseen menevä tehtävä siirrettiin backlogilla In progress-tilaan ja osoitettiin tehtävälle tekijä. Kehittäjä suoritti tehtävään liittyviä alatehtäviä, ja saatuaan suorituksen valmiiksi commitoi tehdyn osan gitin avulla versiohallintaan.

Versionhallinnassa käytäntönä oli commitoida tehty tehtävä uuteen koodihaaraan (branch) versiohallinnassa. Uusi haara nimettiin Jiran tehtävätunnisteen mukaisesti, ja selitteeksi tuli alatehtävän tunniste, sekä mitä muutoksia koodiin tehtiin.

Kun tehtävän kaikki alatehtävät saatiin valmiiksi, uudesta branchista tehtiin pull request, ja siirrettiin tehtävä Jirassa In review-tilaan. Asiakasyrityksen jäsen teki pull requestille koodikatselmoinnin, jossa koodi tarkistettiin ja mahdolliset viat käytiin läpi ja korjattiin.

Kun koodi todennettiin toimivaksi, se hyväksyttiin ja liitettiin (merge) versionhallinnan master- eli päähaaraan, jossa on koodin pääversio, ja joka toimii alkutilanteena kaikille uusille muutoksille. Tehtävän valmistuttua se siirrettiin Jirassa Done-tilaan, jolloin tehtävälle määritetyt pisteet merkittiin suoritetuiksi, ja kehittäjä siirtyi eteenpäin seuraavaan tehtävään.

Projektin alkaessa tutkittiin ja testattiin muutamaa eri CI/CD palvelua. Nämä olivat Atlasianin tarjoama BitBucket Pipelines ja Microsoftin Azure Pipelines. CI/CD palvelua oli tarkoitus alkaa käyttää heti sovelluskehityksen alusta, jotta julkaisuun liittyvät operaatiot helpottuisivat.

Bitbucketin tarjoama vaihtoehto vaikutti aluksi hyvältä, sillä versionhallinta sijaitsi Bitbucketissa, joten oletettiin, että työkalut integroituisivat helposti. Testatessa palvelua huomattiin kuitenkin, että se oli vaikea käyttää ja siihen liittyvä dokumentaatio oli huonolaatuista.

Azure Pipelines oli ennestään tuttu ja todettu helppokäyttöiseksi, joten sen käyttöönottoa BitBucketin versionhallinnan kanssa testattiin. Työkalu todettiin toimivaksi ja tarkoituksenmukaiseksi, ja se otettiin käyttöön projektissa.

Uudet muutokset julkaistiin automaattisesti Azure App Serviceen Azure Pipelinen avulla. Pipeline toimii synkronoidusti BitBucketin versionhallinnan kanssa siten, että Pipeline valvoo versionhallinnan valittua branchia, esimerkiksi development. Kun development haaraan tehdään push, jossa on uudet koodimuutokset, Azuren Pipeline käynnistyy, buildaa eli paketoii ja rakentaa uuden koodin ja julkaisee sen Pipelinen konfiguraatiossa osoitettuun App Serviceen.

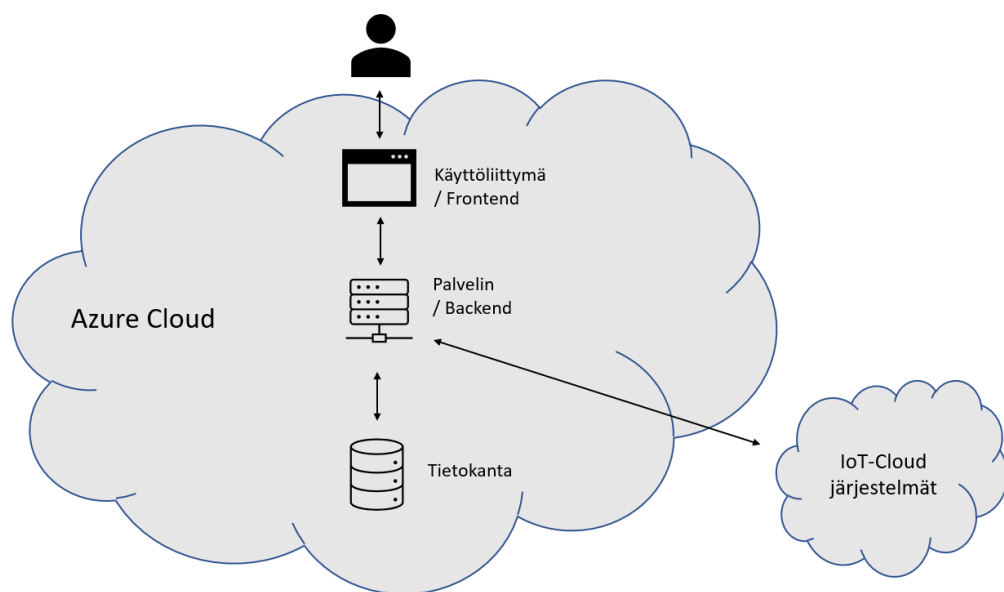
Backend kehityksessä API-rajapinnan toiminnot testattiin ja todennettiin toimivaksi Postmanin avulla. Postmanin käyttö auttaa sovelluskehityksessä ja on hyvä toimintatapa, joka parhaassa tapauksessa nopeuttaa koko projektia.

Sovelluksen avulla backendin toiminnallisuuden testaamisen voi toteuttaa erillään frontendistä, joka mahdollistaa sen, että palvelimen ja käyttöliittymän kehitys voivat edetä eri tahtiin toisiinsa nähden. Kun backend ja frontend jossakin sovelluskehityksen vaiheessa

yhdistetään, Postmanilla testatut toiminnallisuudet tuovat varmuutta järjestelmän toiminnasta.

Kuvassa 13 on esitetty projektissa toteutetun järjestelmän arkkitehtuuri. Tietokanta on SQL-relaatiotietokanta, joka on julkaistu Azuren pilvipalveluun Azure SQL-tietokantana. Backend-sovellus toimii rajapintana tietokannan ja käyttöliittymän välissä, välittäen käyttöliittymältä tulevat HTTP-pyynnöt tietokantaan, joka toteuttaa pyynnön mukaisen operaation. Backend-sovellus hoitaa myös IoT-järjestelmiin kohdistuvat pyynnot.

Sekä backend-sovellus, että käyttöliittymäsovellus ovat julkaistu Azuressa App Service palveluihin ja ne toimivat pilviympäristössä.

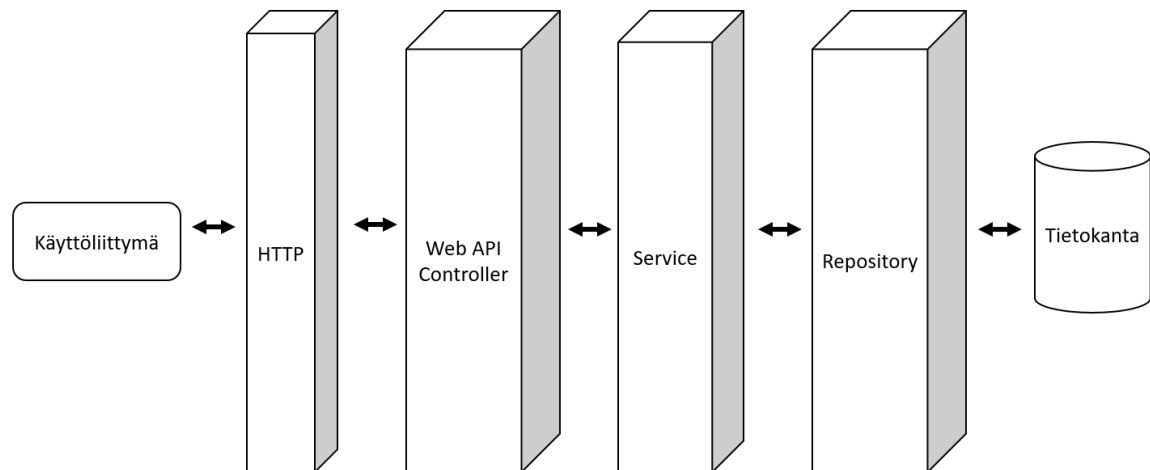


Kuva 13. Projektin arkkitehtuuri

Kuvassa 14 esitetään projektin backend-sovelluksen rakenne. Kuvassa käyttöliittymäsovellus ottaa HTTP-pyyntöjä yhteyttä Web API-sovelluksen Controller kerrokseen, joka on sovelluksen uloin kerros ja tarjoaa API-rajapinnan ulospäin. Controller kerros välittää pyynnön sisällön Service kerrokseen, missä tapahtuu sovelluksen bisneslogiikka, eli sovelluksen "älykkyys". Lopuksi Repository kerros hoitaa asioinnin tietokannan kanssa, eli tekee käyttöliittymän pyytämän operaation ja palauttaa halutun vastauksen.

Clean arkkitehtuurin mukaisesti projektin Repository taso on ulommaisoin kerros, joka tarjoaa yhteyden ulkoiseen tietokantaan. Uloimmalla kerroksella toimii myös Web Api Projekti sekä .NET Framework, jotka suorittavat ohjelman. Controller taso on seuraava kerros, johon ulkoinen käyttöliittymä ottaa yhteyttä. Use cases kerros kuvaa projektin Service tasoa. Entities kerros kuvaa koko sovellukseen vaikuttavia rakenteita.

IoT-järjestelmien toimintaan tutustuttiin niiden tarjoamista dokumentaatioista ja toimintoja testattiin ensin Postmanin avulla. Kun rajapintojen toiminnasta oli testaamalla kerätty tarpeeksi tietoa, niiden hallintaan lisättiin toteutusta backend-sovelluksen Service tasolle.



Kuva 14. Backend-sovelluksen rakenne

6 Pohdinta

Projekti lähti heti alusta etenemään hyvää vauhtia, projektin alussa tehdyn yksityiskohtaisen ja monista näkökulmista toteutetun suunnittelun takia. Vaatimusmäärittely sekä käyttöliittymän ja rajapintojen suunnittelu auttoivat hahmottamaan järjestelmän vaatimat eri kokonaisuudet, ja erityisesti käytettävyyden kannalta mietitty suunnittelu auttoi sekä käyttöliittymän että backend puolen toteutuksessa. Kun eri osat olivat hyvin eritelty ja määritelty, ne oli helppo jakaa edelleen pienemmiksi toteutettaviksi kokonaisuuksiksi.

Suunnitteluvaihe oli jälkeensä todettuna erittäin opettavainen ja äärimmäisen hyödyllinen sovelluskehityksen kannalta. Suunnittelun toteuttaminen vaati paljon energiaa ja keskittymistä, koska vaatimusmäärittelyä, rajapintojen suunnittelua ja käyttöliittymäsuunnittelua tehtiin samanaikaisesti. Tämän vuoksi projektin alkuvaiheessa oli paljon kokouksia, mikä oli paikoitellen puuduttavaa. Projektin edetessä kokouksiin kuitenkin tottui ja niistä tuli osa työskentelyn rutiinia.

Rajapinnan dokumentointiin ja kuvaamiseen tehtiin selvitystyötä muutaman työpäivän verran. Rajapinnan dokumentointiin ja kuvaukseen etsittiin sopivaa työkalua kymmenistä eri vaihtoehdoista. Kolme ylivoimaisesti suosituinta ja kattavinta API-työkalua olivat Postman, Swagger ja Stoplight, jotka toimintojensa ansiosta karsi muut työkalut pois. Lopulliseksi työkaluksi näistä kolmesta työkalusta valikoitui Postman, joka oli yksinkertaisin käytettävyydeltään ja sisälsi kattavimman dokumentaation. Rajapinnan kuvauksen yksinkertaisuus, kattava dokumentaatio ongelmatilanteisiin, sekä sovelluksen käyttöön ja hyvännäköinen, sekä helposti muodostettava dokumentaatio API-pyyntöistä oli suurin syy Postmanin valintaan API-työkaluksi.

Ensimmäinen toteutettu osa oli parhaiten käyty läpi ja siitä oli paras ymmärrys, minkä takia sen toteuttaminen oli nopeaa ja eteni ilman ongelmia. Tästä tuli sekä positiivisia, että negatiivisia seurauksia. Positiivisia olivat muun muassa onnistumisista saatu itsevarmuus ja ruutiini, työ tuntui helpolta ja lähti hyvin käyntiin, mikä aiheutti harhakuvan, että sovelluskehityksen työtahti olisi yhtä nopea koko projektin ajan.

Jokaista järjestelmän viidestä eri toiminnallisesta osa-alueesta ei luonnollisesti ehditty alussa yksityiskohtaisesti suunnittelemaan ja jakamaan tehtäviksi projektinhallintaan, minkä vuoksi näiden osa-alueiden aloituksessa kehitystyö hidastui, koska yksityiskohtaisempaan osa-alueen läpikäymiseen ja suunnitteluun jouduttiin varaamaan aikaa sprinteistä.

Jälkeenpäin ajateltuna oli luonnollista, että muistakin järjestelmän osista piti kasvattaa ymmärrystä ja toteuttaa yksityiskohtaisempaa suunnittelua, mutta tapahtumahetkellä työtahdin hidastuminen tuntui dramaattiselta ja isolta ongelmalta. Ongelma onneksi selvisi helposti

käyttämällä aikaa osa-alueiden läpikäymiseen ja suunnitteluun, ja työtahti palautui nopeasti.

Scrum-viitekehityksessä toteutettu projektinhallinta oli projektin kaikille osapuolille ennestään tuttu, mikä auttoi työskentelyn rytmin löytymisessä. Osapuolet osasivat toimia omissa rooleissaan luontevasti ja työskentely oli ammattimaista.

Sprintit määrittyivät nopeasti sopivan kokoisiksi tehtävien pisteytyksellä ja Scrumin mukaisesti toteutetut sprint review tilaisuudet osoittautuivat hyväksi käytännöksi tulosten läpikäymiseen, tilanteen tarkasteluun ja tiimityöskentelyn reflektointiin.

Projektin kannalta suurin virhe tapahtui API suunnittelussa, johon valittu työkalu ei osoittautunut parhaimmaksi mahdolliseksi helposti luettavan ja muuhunkin kuin sovelluskehitykseen tarkoitettujen rajapintakuvausten luomiseen. Ongelma ei kuitenkaan onneksi vaikuttanut sovelluskehitykseen tai muihin prosesseihin.

Järjestelmän kehittäminen vaati jatkuvaa uuden oppimista ja kaikista sovelluskehitykseen liittyvistä asioista tuli paljon uusia asioita, jotka vaativat opiskelua, mutta koska uudet asiat tulivat pienissä osissa projektin kuluessa eivätkä kerralla, ne eivät tuottaneet vaikeuksia.

Tärkeimpinä oppeina projektista olivat suunnittelun rooli sovelluskehityksessä. Mitä paremmin suunnittelun toteuttaa, sen helpompaa sovelluskehitys on. Kokonaisuudessaan opin näytetyön toteuttaminen lujitti ja kasvatti tekijöiden ammattitaitoa.

7 Jatkokehitys

Sisältö on poistettu asiakkaan pyynnöstä.

8 Yhteenveto

Tässä opinnäytetyössä oli tarkoituksena toteuttaa yritykselle minimivaatimukset täyttävä sovellus, joka auttaisi yritystä kehittämään sen prosesseja laite- ja asiakkuudenhallinnan osalta. Opinnäytetyön tarkoituksena oli myös osoittaa opinnäytetyön tekijöiden osaaminen sovelluskehityksen ja erilaisten kehitystyöhön liittyvien teknologioiden osalta.

Tämä opinnäytetyö toteutettiin toiminnallisena opinnäytetyönä. Opinnäytetyössä lähdettiin liikkeelle toimeksiantajan ongelman esittämisestä, sekä sovellukseen liittyvistä vaatimuksista ja sovelluksen kehitystyön jakautumisesta ohjelmistokehittäjien kesken. Tämän jälkeen perehdyttiin ketterään ohjelmistokehitykseen, sekä sovelluksen suunnitteluun ja sen tuomiin hyötyihin ohjelmistokehityksen kannalta. Tämän jälkeen esiteltiin sovelluksen kehitykseen käytettyjä teknologioita ja sovelluksen arkkitehtuuria. Lopuksi käytiin läpi opinnäytetyön pääasia eli itse projektin toteutus ja sen etenemiseen liittyvä kehitystyö.

Opinnäytetyön tavoitteena oli suunnitella ja toteuttaa hallintasovellus, joka soveltuisi toimeksiantajan asettamiin vaatimuksiin. Sovelluksen kautta tuli pystyä hallinnoimaan sekä asiakkaisiin että IoT-laitteisiin liittyviä tietoja.

Koko järjestelmä oli erittäin laaja ja koostui monista eri osa-alueista. Hyvä suunnittelu alkuvaiheessa auttoi tarkentamaan eri osiin liittyvät tehtävät rajatuiksi kokonaisuuksiksi, jotka oli helppo jakaa toteutettaviksi osatehtäviksi. Järjestelmällisesti toteutettu projektinhallinta teki osatehtävien suorittamisesta nopeaa ja ammattimaista, jonka ansiosta projekti saatiin aikataulun mukaisesti valmiiksi. Kaikki vaatimusmäärittelyssä määritetyt kriittiset ominaisuudet saatiin pääosin valmiiksi ja sovellus eteni projektin päätyttyä testausvaiheeseen.

Toteutuneen sovelluksen osalta päästiin tavoitteisiin ja toimeksiantaja oli tyytyväinen lopputulokseen. Sovelluksen minimivaatimukset täytyivät ja opinnäytetyön osalta riittävän suuri sovelluskokonaisuus saatiin toteutettua. Sovellus itsessään vaatii kuitenkin vielä kehitystyötä, jotta siitä saatava kaikki potentiaalinen hyöty saadaan käyttöön. Voidaan siis sanoa, että tämän opinnäytetyön osuus oli pieni osa suurempaa kokonaisuutta, mutta merkittävä osa perustoinnallisuuksia, jotka ovat välttämättömiä sovelluksen toiminnan kannalta.

Lähteet

Agile Alliance, 2020. What is Agile Software Development? Viitattu 7.10.2020. Saatavissa <https://www.agilealliance.org/agile101/>

Agile Manifesto. Julistuksen takana olevat periaatteet. Viitattu 11.11.2020. Saatavissa <https://agilemanifesto.org/iso/fi/principles.html>

Angular, 2020. Introduction to Angular concepts. Viitattu 21.10.2020. Saatavissa <https://angular.io/guide/architecture>

Angular University, 2020. Angular Single Page Applications (SPA): What are the Benefits? Viitattu 21.10.2020. Saatavissa <https://blog.angular-university.io/why-a-single-page-application-what-are-the-benefits-what-is-a-spa/>

Atlassian, 2018. BitBucket: What is BitBucket? Viitattu 18.10.2020. Saatavissa <https://confluence.atlassian.com/confeval/development-tools-evaluator-resources/bitbucket/bitbucket-what-is-bitbucket>

Atlassian, 2020a. Confluence: Features & Functions. Viitattu 9.10.2020. Saatavissa <https://confluence.atlassian.com/confeval/confluence-evaluator-resources/confluence-features-functions>

Atlassian, 2020b. Set up your site and spaces. Viitattu 21.10.2020. Saatavissa <https://www.atlassian.com/software/confluence/guides/get-started/set-up#step-1>

Atlassian, 2020c. What is Scrum? Viitattu 9.10.2020. Saatavissa <https://www.atlassian.com/agile/scrum>

Atlassian, 2020d. What is a product backlog? Viitattu 23.10.2020. Saatavissa <https://www.atlassian.com/agile/scrum/backlogs>

Atlassian, 2020e. What are the three Scrum roles? Viitattu 23.10.2020. Saatavissa <https://www.atlassian.com/agile/scrum/roles>

Atlassian, 2020f. What are sprints? Viitattu 23.10.2020. Saatavissa <https://www.atlassian.com/agile/scrum/sprints>

Atlassian, 2020g. What is sprint planning? Viitattu 23.10.2020. Saatavissa <https://www.atlassian.com/agile/scrum/sprint-planning>

Atlassian, 2020h. Story points and estimation. Viitattu 23.10.2020. Saatavissa <https://www.atlassian.com/agile/project-management/estimation>

Atlassian, 2020i. What is version control. Viitattu 6.11.2020. Saatavissa <https://www.atlassian.com/git/tutorials/what-is-version-control>

Codecademy, 2020a. What is REST? Viitattu 9.10.2020. Saatavissa <https://www.codecademy.com/articles/what-is-rest>

Codecademy, 2020b. What is a Relational Database Management System? Viitattu 6.11.2020. Saatavissa <https://www.codecademy.com/articles/what-is-rdbms-sql>

Hackernoon, 2017. MVC vs. MVVM: How a Website Communicates With Its Data Models. Viitattu 29.10.2020. Saatavissa <https://hackernoon.com/mvc-vs-mvvm-how-a-website-communicates-with-its-data-models-18553877bf7d>

Interaction Design Foundation, 2002. User Centered Design. Viitattu 14.10.2020. Saatavissa <https://www.interaction-design.org/literature/topics/user-centered-design>

JetBrains WebStorm, 2020. The smartest JavaScript IDE. Viitattu 22.10.2020 Saatavissa <https://www.jetbrains.com/webstorm/>

Ketterä käsikirja, 2015. Esimerkki: Scrum. Viitattu 28.10.2020. Saatavissa https://tech.utu.fi/embedded_kasikirja/1/1/index.html

Medium, 2019. Why Angular is your best choice for your next projects? Viitattu 29.10.2020. Saatavissa <https://medium.com/@maaouikimo/why-angular-is-your-best-choice-for-your-next-projects-9d754fb18f91#:~:text=Angular%20framework%20is%20embedded%20with,code%20that%20could%20unite%20them>

Microsoft, 2017d. The Model-View-ViewModel Pattern. Viitattu 28.10.2020. Saatavissa <https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm>

Microsoft, 2019a. What is SQL Server Management Studio (SSMS)? Viitattu 7.10.2020. Saatavissa <https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver15>

Microsoft, 2019b. An introduction to NuGet. Viitattu 14.10.2020. Saatavissa <https://docs.microsoft.com/en-us/nuget/what-is-nuget>

Microsoft, 2019c. Welcome to the Visual Studio IDE. Viitattu 14.10.2020. Saatavissa <https://docs.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2019>

Microsoft, 2019d. What is Azure Pipelines? Viitattu 7.10.2020. Saatavissa <https://docs.microsoft.com/en-gb/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops>

Microsoft, 2020a. Introduction to .NET. Viitattu 7.10.2020. Saatavissa <https://docs.microsoft.com/en-us/dotnet/core/introduction>

Microsoft, 2020b. A tour of the C# language. Viitattu 7.10.2020. Saatavissa <https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>

Microsoft, 2020c. Create web APIs with ASP.NET Core. Viitattu 14.10.2020. Saatavissa <https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-3.1>

Microsoft, 2020d. App Service overview. Viitattu 7.10.2020. Saatavissa <https://docs.microsoft.com/fi-fi/azure/app-service/overview>

Microsoft, 2020e. What is Azure Active Directory? Viitattu 21.10.2020. Saatavissa <https://docs.microsoft.com/en-us/azure/active-directory/fundamentals/active-directory-whatis>

Network Encyclopedia, 2020. Key in a Relational Database. Viitattu 6.11.2020. Saatavissa <https://networkencyclopedia.com/key-in-a-relational-database/>

OAI, OpenApi Specification. Viitattu 9.10.2020. Saatavissa <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#introduction>

Postman Inc, 2020a. Using the API Builder. Viitattu 9.10.2020. Saatavissa <https://learning.postman.com/docs/designing-and-developing-your-api/the-api-workflow/>

Postman Inc, 2020b. Building requests. Viitattu 22.10.2020. Saatavissa <https://learning.postman.com/docs/sending-requests/requests/>

Postman Inc, 2020c. Grouping requests in collections. Viitattu 6.11.2020. Saatavissa <https://learning.postman.com/docs/sending-requests/intro-to-collections/>

QuadraNet, 2019. What is Git? Git Explained. Viitattu 18.10.2020. Saatavissa <https://blog.quadranet.com/what-is-git-git-explained/>

Red Hat, 2020. What is an API? Viitattu 8.10.2020. Saatavissa <https://www.red-hat.com/en/topics/api/what-are-application-programming-interfaces>

Restfultapi.net, 2020. How to design a REST API. Viitattu 22.10.2020. Saatavissa <https://restfultapi.net/rest-api-design-tutorial-with-example/>

Simform, 2020. What is Node.js? Where, when and how to use it with examples. Viitattu 21.10.2020. Saatavissa <https://www.simform.com/what-is-node-js/>

SmartBear Software, 2020. What Is OpenApi? Viitattu 9.10.2020. Saatavissa <https://swagger.io/docs/specification/about>

Stack Overflow, 2020. Best Practices for REST API design. Viitattu 22.10.2020. Saatavissa <https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/>

Tech, 2015. Esimerkki: Scrum. Viitattu 7.10.2020. Saatavissa https://tech.utu.fi/embedded_kasikirja/1/2/index.html

The Clean Code Blog, 2012. The Clean Architecture. Viitattu 21.10.2020. Saatavissa <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>