Haaga-Helia
ammattikorkeakoulu Oy

Projektin- ja ajanhallintasovellus Kooditar Oy:lle

Henry Kari

Opinnäytetyö
Tietojenkäsittelyn
koulutusohjelma
2018

Haaga-Helia
ammattikorkeakoulu Oy

Tiivistelmä

Tekijä(t)
Henry Kari

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko
Projektin- ja ajanhallintasovellus Kooditar Oy:lle

Sivu- ja liitesivumäärä
43 + 15

Tämän opinnäytetyön tarkoituksena oli tuottaa vuonna 2016 perustetulle Kooditar Oy:lle projektin- ja ajanhallintasovelluksen prototyyppiversio ketterän ohjelmistokehitystyön tarpeisiin. Sovellukseen haluttiin yhdistää oleellisimmat ominaisuudet Trello- ja Toggl-sovelluksista, joita yritys on aikasemmin käyttänyt.

Opinnäytetyön teoriaosuuksissa käytiin läpi ketterän ohjelmistokehityksen periaatteita kiinnittäen erityisesti huomiota työnkulun visualisointiin. Myös erilaisia olemassaolevia projektinhallintasovelluksia esiteltiin lyhyesti.

Sovellusta kehitettiin ohjelmistokehityksen elinkaarenmukaisissa vaiheissa iteratiivisin menetelmin. Kehityksen vaiheisiin kuuluvat määrittely, suunnittelu, toteutus, testaus ja julkaisu.

Sovelluksen teknologioiksi valittiin palvelinohjelmoinnin osalta .NET Core -viitekehys ja C#-ohjelmointikieli. Käyttöliittymän toteutustavaksi puolestaan valittiin Angular-alusta, joka hyödyntää TypeScript-, HTML- ja CSS-kieliä. Palvelimella ajettava sovellus toteuttaa REST-rajapinnan, johon käyttöliittymäsovellus ottaa yhteyden.

Opinnäytetyön puitteissa sovelluksesta saatiin valmiiksi REST-rajapinta, joka on julkaistu Amazonin AWS-palvelimella. Käyttölittymäsovelluksesta saatiin valmiiksi runko, joka ottaa yhteyden REST-sovellukseen. Käyttöliittymän kehityksestä valmiiksi päätetään myöhempänä ajankohtana Kooditar Oy:n tiimissä.

Asiasanat
ohjelmistokehitys, ketterät menetelmät, projektinhallinta, ajanhallinta

Sisällys

1 Johdanto

Kooditar Oy on vuonna 2016 perustettu ohjelmistoalan yritys. Koodittaren liikevaihto vuosina 2016 ja 2017 koostui yksinomaan tilaustyönä tehdyistä asiakasprojekteista. Koodittaren perustajien haaveena on kuitenkin alusta asti ollut valmistaa yritykselle jotain omaa: jokin tuote, jota myydä eteenpäin. Ideoita omalle tuotteelle on Kooditar-tiimissä esitetty runsaasti, ja osaa niistä on lähdetty toteuttamaankin. Keväällä 2017 Kooditar julkaisi Fyrkkaa!-mobiilisovelluksen, jonka avulla käyttäjä löytää lähimmät Otto-pankkiautomaatit laitteensa kompassiominaisuutta hyödyntäen. Tuotteelle ei ole kuitenkaan onnistuttu tähän mennessä keksimään minkäänlaista ansaintamallia, ja se onkin toistaiseksi jäänyt eräänlaiseksi työnäytteeksi.

Yksi esitetyistä tuoteideoista on ollut oma projektinhallintasovellus, joka helpottaisi tiimityöskentelyä kanban- ja scrum-tyylisissä ketterissä ohjelmistokehitysprojekteissa. Sovellus itsessään olisi hyödyllinen työkalu Koodittarelle, sillä se tehostaisi työskentelyä yrityksen nykyisten ja tulevien projektien parissa. Siten sen hyödyntäminen kaupallisena tuotteena ei ole välttämätöntä, vaikka siitä joskus sellainen haluttaisiin vielä jalostaa. Tämän opinnäytetyön tarkoituksena on tuottaa prototyyppiversio edellä kuvatusta sovelluksesta.

Kooditar on aikaisemmissa projekteissaan hyödyntänyt Trello- ja Toggl-sovelluksia. Trello on monipuolinen ja joustava työkalu, jota voi soveltaa erilaisiin projekteihin useilla eri toimialoilla. Ohjelmistokehitysprojekteissa sitä voi käyttää esimerkiksi kanban-taulun luomiseen ja muokkaamiseen. Toggl puolestaan on työajanseurantasovellus, jolla voi pitää kirjaa työtunneista joko reaaliaikaisesti tai syöttämällä tunnit manuaalisesti. Koodittaren omalla projektinhallintasovelluksella halutaan yhdistää nämä ominaisuudet, jotta projektinhallinta voidaan hoitaa yhdellä sovelluksella useamman sijaan. Oma projektinhallintatyökalu mahdollistaa myös sen, ettei Koodittaren tarvitse siirtyä käyttämään kaupallisten tuotteiden maksullisia versioita.

2 Ketterä ohjelmistokehitys

Ketterä kehitys (Agile development) on kattotermi useille erilaisille projektinhallinnan viitekehyksille, joilla pyritään vastaamaan asiakkaan vaatimuksiin, jotka muuttuvat nopeasti. Ketterät menetelmät ovat yleistyneet viime vuosikymmeninä, ja niistä on 2000-luvun aikana muodostunut alan standardi. Yleisimmin käytössä olevia ketteriä ohjelistokehityksen projektinhallintaviitekehyksiä ovat Kanban ja Scrum (The Clever PM, 2016). Tässä luvussa käsitellään ketteriä kehitysmenetelmiä erityisesti työnkulun ja sen visualisoinnin näkökulmasta, sillä ne ovat oleellisimpia tässä opinnäytetyössä kehitettävän projektinhallintasovelluksen kannalta.

2.1 Kanban

Kanban on alunperin autoteollisuuden tarpeisiin kehitetty käsite. Se kehitettiin Toyotalla liukuhihnojen aikataulutusjärjestelmäksi. Sittemmin Kanbanista on muodostunut toimintamalli, joka on käytössä useilla eri toimialoilla. (Cole & Scotcher, 2015, s. 69-70)

Kanbanin periaatteisiin kuuluu mm. työnkulun visualisointi ja käynnissä olevan työmäärän rajoittaminen. Kanbanin keskeisin työkalu on kanban-taulu, jonka voi yksinkertaistetusti mieltää eräänlaiseksi virtuaaliseksi tehtävälistaksi. Yksinkertaisimmillaan kanban-taulu sisältää kolme saraketta, jotka kuvastavat eri työvaiheita: töitä, joita ei ole vielä aloitettu (to do); töitä, jotka ovat käynnissä (in progress) ja töitä, jotka ovat valmistuneet (done). Taulun sarakkeiden nimeämiskäytännöt vaihtelevat ja taulu voi sisältää myös useampia työvaiheita kuin edellä luetellut. Ohjelmistokehitysprojekteihin sovellettuna kanban-taulu voi sisältää myös alalle ominaisia työvaiheita kuten testausvaiheen. Taulun voi piirtää esimerkiksi valkotaululle, jossa yksittäisiä työtehtäviä voi kuvastaa tarralapuilla, joita siirretään sarakkeesta toiseen työn edetessä (kuva 1). (Cole & Scotcher, 2015, s. 71-76).

Kuva 1: yksinkertainen kanban-taulu valkotaululla (Wikimedia Commons)

2.2 Scrum

Scrum on Kanbania selkeämmin ohjelmistokehitysprojekteja varten suunniteltu viitekehys. Scrumissa myös aikataulut ja työrytmi ovat täsmällisemmin määriteltyjä. Scrumissa ohjelmistokehitysprojekti on jaettu määrätyn ajan kestäviin työskentelyjaksoihin eli sprintteihin (Cole & Scotcher, 2015, s. 87). Sprintti käynnistyy suunnittelupalaverista, jossa päätetään, mitä ominaisuuksia tuotteen kehitysjonosta (product backlog) valitaan toteutettavaksi sprintin aikana. Normaali kehitystyöpäivä alkaa n. 15 minuutin palaverilla, jossa kukin tiimin jäsen kertoo vuorollaan kolme asiaa: mitä hän teki edellisenä työpäivänä sprintin tavotteiden toteuttamiseksi, mitä hän tekee sen eteen tänään ja millaisia esteitä tai hidasteita sen tielle saattaa tulla. Sprintti päättyy katselmukseen ja retrospektiiviin. Katselmuksessa tarkastellaan sprintin tuotoksia ja retrospektiivissä pohditaan tiimin toimintaa yleisesti ja mietiään siihen parannusehdotuksia seuraavalle sprintille. (Cole & Scotcher, 2015, s. 96-104).

Scrumin työnkulkua voi visualisoida hyvin samankaltaisesti kuin kanbanin. Erona on se, että scrumin tehtävätauluun halutaan vain yhden sprintin työtehtävät kerrallaan. Se on siis tyhjennettävä ja täytettävä uusilla työtehtävillä seuraavaan sprinttiin siirryttäessä.

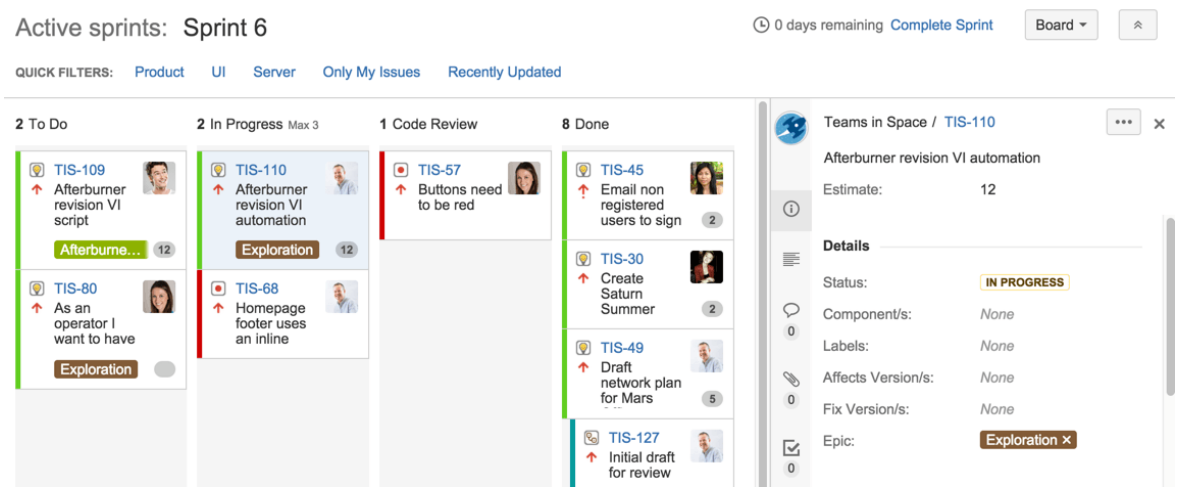
3 Olemassaolevia projektinhallintasovelluksia

Ohjelmistokehitysprojekteissa on usein käytännöllisempää hyödyntää digitaalisia palveluita pelkän valkotaulun sijaan. Projektinhallinnassa se mahdollistaa työtehtävien tilan reaaliaikaisen päivittämisen ja raportoinnin tusseja ja tarralappuja kuluttamatta. Digitaalisesti toteutettua tehtävätaulua työntekijän on myös mahdollista muokata tehdessään töitä etänä. Markkinoilla onkin tarjolla runsaasti erilaisia projektinhallintasovelluksia. Tässä käydään läpi niistä muutamia.

3.1 Jira

Jira on Atlassianin ohjelmisto, joka kehitettiin alun perin vianjäljitysohjelmaksi. Nykyään Jira tarjoaa monipuolisesti työkaluja sekä Scrum- että Kanban-tyylisiin projekteihin. (Li, 2016)

Luodessaan uuden projektin Jirassa käyttäjällä on mahdollisuus valita useista erityyppisistä projektipohjista sopivin. Pohjia on olemassa esimerkiksi Scrum- ja Kanban-projekteille. (Li, 2016)



Kuva 2 Jiran scrum-taulu (Atlassian)

3.2 Trello

Trello on Jiraa yksinkertaisempi, mutta samalla joustavampi projektinhallinnan työkalu. Toisin kuin Jira, joka tarjoaa käyttäjälle valmiin sapluunan kanban- tai scrum-taululle, Trellon taulun sisältö on täysin käyttäjän räätälöitävissä. Käyttäjä luo tauluun listoja, jotka

voivat esimerkiksi olla kanbanin tai scrumin työvaiheita, mutta tämän ominaisuuden hyödyntäminen ei rajoitu ketteriin kehitysprojekteihin tai ohjelmistokehitykseen ylipäänsä. (wpcurve.com, 2015)

3.3 Toggl

Toggl on työtuntien kirjanpitoon tarkoitettu sovellus, jossa käyttäjä voi merkitä työtunnit joko reaaliaikaisesti ajastimella, eli painalla sovelluksen start-painiketta aloittaessaan työt ja pysäyttämällä ajastimen lopettaessaan työt tai mennessään tauolle, tai manuaalisesti syöttämällä työn alkamis- ja päättymisajankohdan kuten muissakin tuntikirjanpitojärjestelmissä. (Pot, 2011).

4 Ohjelmistokehityksen elinkaari

Ohjelmistokehitys on vaiheittainen prosessi, joka käynnistyy projektin määrittelystä johtaen lopulta valmiiseen, julkaistuun tuotteeseen. Kehityksen elinkaari ei yleensä kuitenkaan pääty tähän, vaan se jatkuu usein ylläpito- tai jatkokehitysvaiheeseen. Kehityksen elinkaari voidaan jakaa vaiheisiin usealla eri tavalla, mutta tavallisimpia ohjelmistokehityksen vaiheita ovat määrittely (requirements), suunnittelu (design), toteutus (implementation), testaus (testing), julkaisu (delivery) ja ylläpito (maintentance). Perinteisesti ohjelmistokehitysprojekteja on toteutettu ns. vesiputousmallina, jossa kukin ohjelmistokehitysprosessin vaihe on tarkkaan määritelty. Vesiputousmallissa toteutetaan ensin yksi vaihe loppuun, jonka jälkeen siirrytään seuraavaan.

Ketterissä ohjelmistokehitysmenetelmissä hyödynnetään usein iteratiivisempaa lähestymistapaa. Iteratiivinen malli pitää sisällään täysin samat kehitysvaiheet kuin vesiputousmalli, mutta siinä pyritään hyödyntämään tehokkaammin mahdollisuutta palata aikaisempiin vaiheisiin. (Powell-Morse, 2016).

5 Teknologiat

Tässä opinnäytetyössä tehtävän sovelluksen palvelinohjelmoinnin toteutusalustaksi on valittu Microsoftin .NET Core -viitekehys. .NET Coren kirjastovalikoimasta hyödynnetään Entity Framework-, ASP.NET Identity- ja ASP.NET Web API -kirjastoja. Tietokannan toteutuksesta vastaa SQL Server 2014. Käyttöliittymä toteutetaan Angular-sovelluksena, joka pitää sisällään HTML5:llä, CSS:llä ja JavaScriptiksi kääntyvällä TypeScriptillä toteutettuja komponentteja.

Teknologioiden valinnassa on pyritty huomioimaan mm. niiden modernius, skaalautuvuus ja mobiiliyhteensopivuus. Valitut teknologiat ovat koko tiimillemme oppimiskokemus. Siten niiden käyttöönotto mahdollistaa oman osaamisemme kehittämisen ja ajan hermolla pysymisen yrityksessämme. Samalla tulemme havaitsemaan, millä tavoin ne sopivat omiin käyttötarkoituksiimme ja kannattaako meidän hyödyntää niitä jatkossakin.

5.1 .NET Framework ja .NET Core

.NET on Microsoftin kehittämä ohjelmistoviitekehys. Ensimmäinen versio .NETistä julkaistiin vuonna 2002. .NET-ympäristössä on mahdollista tehdä kehitystyötä useilla eri ohjelmointikielillä, kuten C#, C++, Visual Basic ja F#, siten, ne kaikki käänytyvät yhteiseksi välikieleksi, mikä mahdollistaa sen, että eri ohjelmointikielillä luodut komponentit voivat hyödyntää toisiaan. Entity Framework -kirjasto tarjoaa saumattoman yhteyden .NET-projektin luokkarakenteesta Microsoftin SQL Server -tietokantaan. (Troelsen & Japikse, 2015).

NETin perusversiosta poiketen, jota on sellaisenaan ainoastaan mahdollista ajaa Windows-ympäristössä, .NET Core on alustariippumaton ja avoimen lähdekoodin viitekehys. (Chowdury, 2016)

5.1.1 C# ja oliopohjainen ohjelmointi

C# (C Sharp) on .NET-viitekehyksen puitteissa kehitetty ohjelmointikieli. Se muistuttaa syntaksiltaan läheisesti Javaa, mutta siihen on otettu vaikutteita myös muista ohjelmointikielistä, kuten C++:sta ja Visual Basicistä (VB). Siten se tukee oliopohjaisen ohjelmoinnin lisäksi monia funktionaalisen ohjelmoinnin piirteitä. (Troelsen & Japikse, 2015). Troelsenin ja Japiksen mukaan C#:ssa yhdistyy Javan siistiys, VB:n yksinkertaisuus ja C++:n tehokkuus ja joustavuus.

Muiden oliopohjaisten ohjelmointikielien tavoin C#:lla toteutettu ohjelma koostuu rakenteista, joita kutsutuaan luokiksi (class). Luokkien avulla C#-kehittäjä voi luoda sovellukseen omia tietotyyppejä ja käyttää niiden ilmentymiä eli olioita esimerkiksi muuttujien arvoina. Luokkien tehokas hyödyntäminen mahdollistaa laajempien, paremmin jäsenneltyjen ja helpommin ylläpidettävien sovellusten kehittämisen. (Michaelis, 2013, s. 209-211)

Luokkien hyödyntäminen on luonteva tapa toteuttaa sovelluksen tietomalleja. Luokkiin määritellään muuttujia, eli kenttiä, jotka kuvastavat luokasta johdetun olion tilaa, sekä metodeja, joilla olion tilaa voidaan lukea tai muuttaa. Yleinen käytäntö monissa oliopohjaisissa ohjelmointikielissä, kuten Java, on ollut toteuttaa luokan kentät yksityisinä kenttinä, joihin pääsee käsiksi vain luokan sisältä käsin. Luokkaan sitten määritellään usein varsin yksinkertaisia metodeja, joiden ainoa tehtävä on lukea muuttujan arvo (”getter”-metodit) tai asettaa muuttujalle arvo (”setter”-metodit). Tällaista käytäntöä kutsutaan kapseloinniksi (encapsulation). Kuvassa 3 on Javalla toteutettu yksinkertainen Pizza-luokka, jonka sisäiset tietorakenteet, eli name- ja id-kentät on kapseloitu.

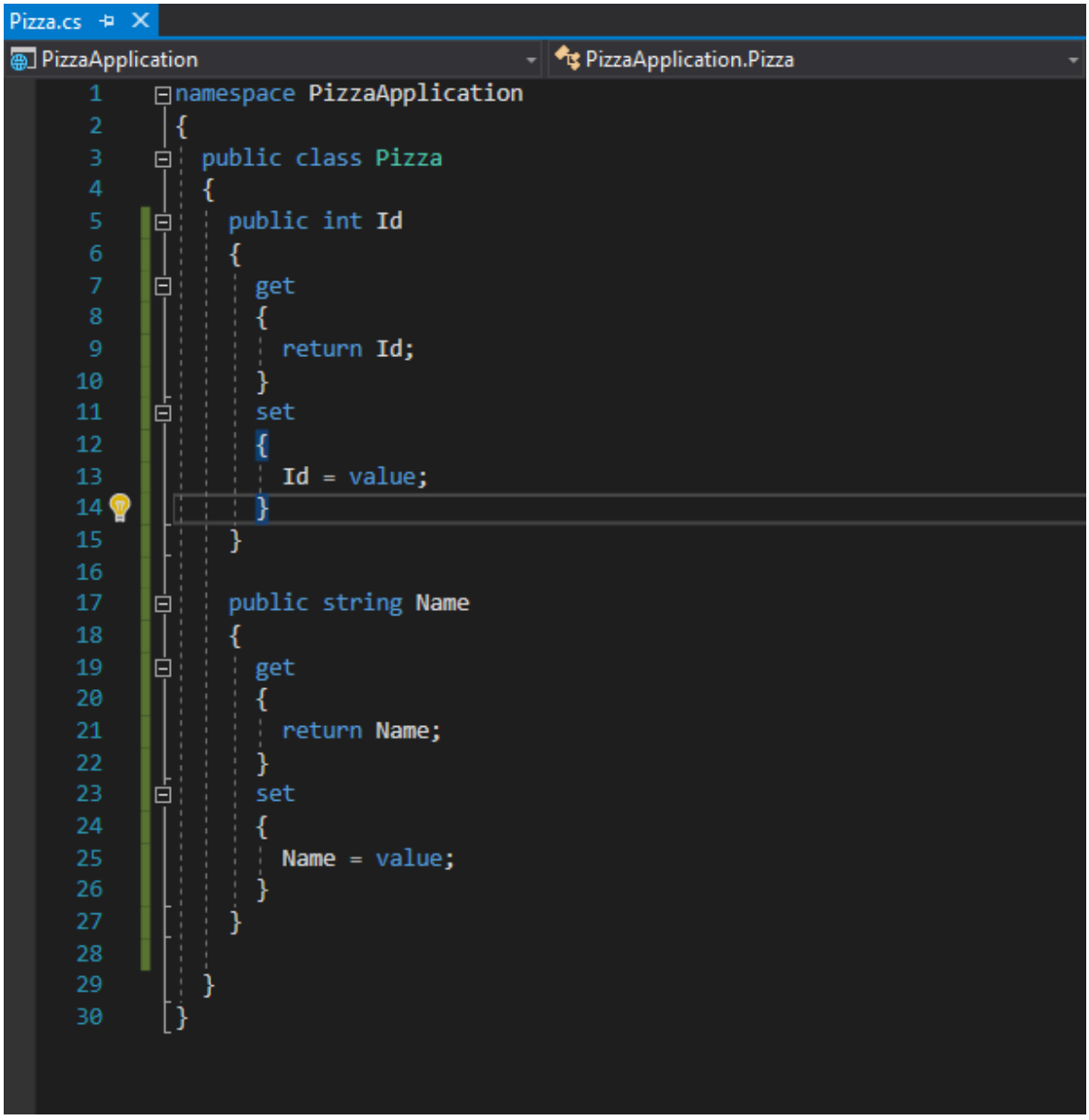
Pizza.java

```
public class Pizza {

    private int id;
    private String name;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

}
```

Kuva 3 Yksinkertainen Pizza-luokka Javalla toteutettuna

Edellinen Java-esimerkki toimisi lähes sellaisenaan täysin validina C#-koodina. Myös C#-luokka voidaan kapseloida määrittelemällä sille yksityisiä kenttiä ja luomalla niille getter- ja setter-metodeita. C#:n käytäntöjen mukaisesti näin ei kuitenkaan usein tehdä, vaan

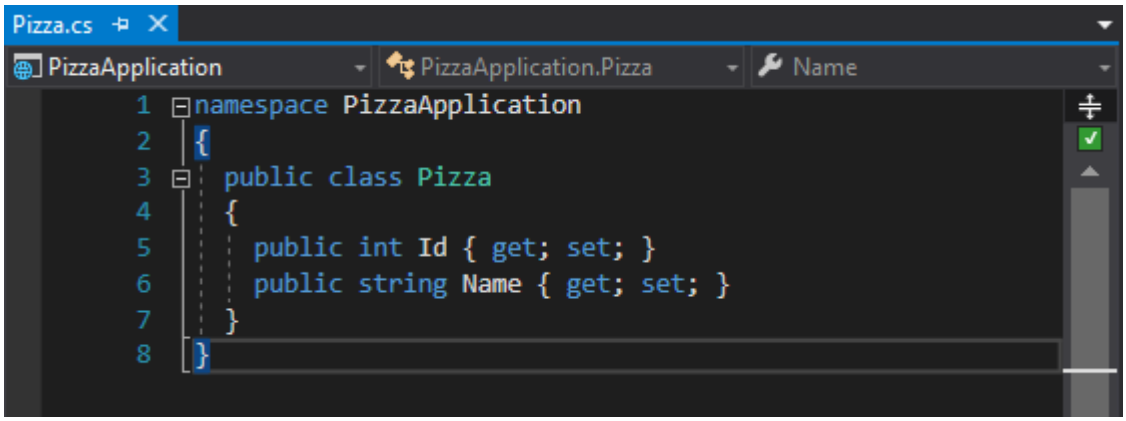
C#:ssa on tavallisempaa hyödyntää nk. property-syntaksia, joka tekee koodista huomattavasti kompaktimpaa. Kun kentät määritellään property-tyylisesti, niitä pystytään käsittelemään muista luokista käsin ikään kuin ne olisivat julkisia kenttiä. Tämä vähentää tarvetta käyttää koodissa metodikutsuja yksinkertaisissa arvojen luku- tai asetusoperaatioissa. (Michaelis, 2013, s. 230.) Kuvassa 4 on edellinen pizzaesimerkki C#:lla toteutettuna property-syntaksia hyödyntäen. Getterit ja setterit on toteutettu koodilohkoina välittömästi kentän määrittelyn yhteydessä.



Kuva 4 Pizza-luokka property-syntaksia hyödyntäen

Tätäkin ohjelmointikäytännettä on haluttu tiivistää entisestään. C#:n versioon 3.0 päätettiin toteuttaa property-kenttien getterit ja setterit automaattisesti implementoituina johtuen siitä, miten yleisesti niitä käytetään ja miten triviaaleja niiden toiminnot useimmiten ovat.

(Michaelis, 2013, s.232.) Kuvassa 5 on toteutettu edellinen esimerkki, jossa hyödynnetään lyhyempää property-syntaksia. Property-kentissä ei tarvitse kertoa, mitä getterit ja setterit tekevät, vaan ainoastaan määritellään niiden olemassaolo. Mikäli gettereihin tai settereihin halutaan toteuttaa jotakin monimutkaisempaa logiikkaa kuin yksinkertaisia arvon luku- ja asettamisoperaatioita, tätä lähestymistapaa ei silloin voida hyödyntää.



Kuva 5 Pizza-luokka, joka hyödyntää lyhennettyä versiota property-syntaksista

C# tukee muiden .NET-kielien tavoin LINQ-toiminnallisuuksia (Language Integrated Query). LINQ mahdollistaa SQL-tyylisten kyselyiden tekemisen itse ohjelmakoodissa. Kyselyiden kohteet voivat olla lähes mitä tahansa joukkoja, kuten listoja, taulukoita tai tietokantaobjektijoukkoja. Ilman LINQ:iä oliojoukkojen käsittely halutulla tavalla edellyttäisi ylimääräisiä for-silmukoita, mikä lisäisi koodin pituutta ja kompleksisuutta. (Troelsen & Japikse, 2015). Entity Framework-työkalun avulla LINQ-kysely voidaan kääntää ohjelman ajon aikana automaattisesti SQL-kyselyksi (Barskiy, 2015). Tämä tekee sovelluksen keskustelun SQL-tietokannan kanssa saumattomaksi.

5.1.2 Entity Framework

Lähes kaikki ohjelmistot joutuvat varastoimaan käyttämäänsä dataa jollain tavalla. Yksi tavallisimpia ratkaisuja siihen on SQL-kielellä toteutettu relaatiotietokanta. Perinteisesti kommunikointi SQL-tietokannan kanssa tapahtui ohjelmistokoodiin upotetuilla SQL-lauseilla. Haluttu data täytyi sitten lukea SQL-kyselyn tuloksesta ja asettaa muuttujien tai olioiden kenttien arvoiksi. (Barskiy, 2015)

Koska edellä kuvattu menettely on osoittautunut sekä työlääksi että epävapaaksi, useisiin eri ohjelmointikieliin on kehitetty työkaluja sujuvoittamaan sovelluksen ja tietokannan välistä kommunikointia. Näistä työkaluista käytetään nimitystä *object-relational mapping*, ORM. ORM:t mahdollistavat tietokannan taulujen käsittelyn ohjelmointikielen natiiveina

olioina. Siten kehittäjä pystyy käsittelemään tietokannan sisältöä ohjelmakoodista käsin tarvitsematta kirjoittaa lainkaan SQL:ää. (Barskiy, 2015) .NET Frameworkiin on versiosta 3.5 (Service Pack 1) lähtien kuulunut Entity Framework-niminen ORM-kirjasto (Troelsen & Japikse, 2015).

Entity Frameworkin ensimmäinen versio tuki ainoastaan Database-First-lähestymistapaa. Ensin täytyi suunnitella ja luoda tietokanta, josta voitiin sitten generoida C#-koodiin tietokantaa mallintava luokkarakenne. Versiossa 4.1 toteutettiin Code-First-menetelmä, joka mahdollisti sen, että kehittäjä voi ensin luoda C#:n luokkarakenteen, jonka perusteella luodaan tietokanta automaattisesti. (Barskiy, 2015)

Entity Frameworkia hyödyntävä projekti pääsee käsiksi tietokantadataan DbContext-luokasta johdetusta aliluokasta käsin. DbContextissa on julkisina kenttinä joukko DbSet<T> -tyyppisiä muuttujia, joissa T on jotakin tietokannan taulua kuvastava tyyppi. Alla olevassa esimerkissä (kuva 6) on yksinkertaisen pizzasovelluksen DbContext-luokan toteutus, jossa on kaksi tietokantajoukkoa: pizzat ja täytteet (topping). Luodun kontekstin ja siihen kuuluvien DbSet-kenttien avulla Entity Framework osaa luoda tietokantaan Pizzas- ja Toppings-nimiset taulut edellyttäen, että projektiin on määritelty toimiva tietokantayhteys.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.EntityFrameworkCore;
6
7  namespace PizzaApplication
8  {
9      public class PizzaContext : DbContext
10     {
11         public DbSet<Pizza> Pizzas { get; set; }
12         public DbSet<Topping> Toppings { get; set; }
13     }
14 }
15
```

Kuva 6 Esimerkki DbContext-luokan toteutuksesta

Kun DbContext-toteutus on luotu, sitä pystytään hyödyntämään muualla sovelluksessa. Sovellus pystyy sen avulla hakemaan dataa tietokannasta ja asettamaan haettu rivi tai joukko rivejä muuttujan arvoksi. Kuvassa 7 on WebAPI:n mukainen REST-rajapinnan toteuttava Controller-luokka PizzasController. (WebAPI:a ja RESTiä käsitellään tarkemmin

seuraavassa luvussa.) Tietokantakonteksti injektoidaan luokkaan sen konstruktorissa. Metodi Get(int id) hakee tietokannasta pizza-olion parametrina annetun id:n perusteella ja palauttaa sen JSON-muodossa client-ohjelmalle HTTP-kutsun yhteydessä. LINQ-kysely ”var pizza = _pizzaContext.First(p => p.Id == id);” toteuttaa SQL-kyselyn. Jos metodia kutsutaan esimerkiksi parametrilla ”5”, Entity Framework ajaa taustalla SQL-lauseen ”SELECT * FROM Pizzas WHERE Id = 5” ja asettaa kyselyn tuloksen paikallisen muuttujan ”pizza” arvoksi.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace PizzaApplication.Controllers
{
    [Route("api/[controller]")]
    public class PizzasController : Controller
    {

        private PizzaContext _pizzaContext;

        public PizzasController(PizzaContext pizzaContext)
        {
            _pizzaContext = pizzaContext;
        }

        // GET api/pizzas/5
        [HttpGet("{id}")]
        public ActionResult Get(int id)
        {
          var pizza = _pizzaContext.Pizzas.First(p => p.Id == id);
          return Ok(pizza);
        }

    }
}
```

Kuva 7 Controller-luokka, jossa hyödynnetään DbContext-luokan toteutusta

Usein controllereiden ei kuitenkaan haluta palauttavan tietokannan dataa sellaisenaan, vaan toisinaan sitä halutaan esimerkiksi yksinkertaistaa datan siirron keventämiseksi tai muusta syystä esittää dataa tietokannan mallista poikkeavalla tavalla. Tällöin sovellukseen luodaan ns. Data Transfer Object -luokkia (DTO), joihin tietokannan taulujen data asetetaan halutusti muokattuna. (Microsoft, 2014). Jos Pizza-luokalle tehtäisiin DTO-luokka, se olisi nimeämiskäytäntöjen mukaisesti nimeltään PizzaDTO. Tuolloin controller-luokasta palautettava olio olisi myös tyypiltään PizzaDTO.

5.1.3 ASP.NET Web API

ASP.NET Web API on .NET-viitekehyksen tapa toteuttaa REST-rajapinta (Representational State Transfer). REST ei sinänsä ole mikään teknologia, vaan arkkitehtuurityyli, jonka määrittävät tietyt rajoitteet. RESTille ominaista on asiakasohjelman (client) ja palvelimella ajetun ohjelman (server) eristäminen toisistaan. REST tarjoaa asiakasohjelmalle resursseja URL-osoitteiden perusteella tarvitsematta ottaa kantaa siihen, millä tavoin resurssien antamaa dataa käsitellään siinä tai miten se on esitetty käyttöliittymässä. Vastaavasti client-puolen ohjelman ei tarvitse välittää siitä, minkälainen sovelluslogiikka toimii taustalla tarjotun datan tuottamiseksi. Tämä mahdollistaa sen, että REST-tyyliseen ohjelmistorajapintaan voidaan päästä käsiksi monilla erilaisilla käyttöliittymätoteutuksilla, kuten web- tai mobiilisovelluksilla. REST toteuttaa tietokannan lukemiseen ja muokkaamiseen tarkoitetut CRUD-operaatiot (Create, Read, Update, Delete) HTTP-protokollan mukaisilla toiminnoilla POST, GET, PUT ja DELETE. (Kanjilal, 2013, s.24-28)

ASP.NET Web APIn taustalla on MVC-malli (model, view, controller) ilman, että palvelin palauttaa käyttäjälle näkymää (view) (Chauhan, 2013). REST-resurssien tarjoaminen käyttäjälle on toteutettu Controller-luokissa, joissa URL-reitit kutsuvat eri metodeja (Kanjilal, 2013, s.160).

5.1.4 Identity-kirjasto ja JSON web token -kirjautuminen

Sovelluksen käyttäjien hallinta on toteutettu Microsoftin Identity-kirjastoa hyödyntäen. TreglContext-luokka, joka kuvastaa sovelluksen tietokantakontekstia, periytyy IdentityDbContext-luokasta, mikä mahdollistaa sen, että Entity Framework osaa luoda tietokantaan kaikki käyttäjien hallintaan vaadittavat tietokantataulut automaattisesti, kuten sovelluksen käyttäjää kuvastava AspNetUsers-taulu, johon tallennetaan mm. käyttäjän käyttäjänimi, sähköpostiosoite ja salasana SHA-256-algoritmilla salatussa muodossa (Liite 2).

Käyttäjän kirjautuminen on toteutettu siten, että kun käyttäjä on lähettänyt käyttäjätunnuksensa ja salasanansa http-pyyntönä palvelimelle, palvelimelta palautuu vastauksena JSON web token (JWT), eli koodattu merkkijono, jossa on tiedot kaikista käyttäjän oikeuksita palveluun. Käyttöliittymäpuolen sovellus tallentaa tuon merkkijonon selaimen muistiin. Kun käyttäjä navigoi sivulla, merkkijono on mukana kaikissa http-

pyynnöissä header-attribuuttina. Http-pyyntöjen yhteydessä palvelimella tarkistetaan, mitä oikeuksia käyttäjän JWT-merkkijonoon on koodattu ja annetaan sen mukaan käyttäjälle palvelun eri resursseja käytettäväksi.

JSON web token on voimassa vain rajoitetun ajan, joten jwt:n vanhennettua käyttäjän on kirjauduttava palveluun uudelleen. Jokaisen kirjautumisen yhteydessä käyttäjälle luodaan uusi jwt.

JSON web token pitää sisällään eräänlaisen allekirjoituksen, jossa on jwt:n tiedot SHA-256-salattuna. Tuon allekirjoituksen avulla pystytään tarkistamaan jwt:n oikeellisuus ja voimassaoloaika, mikä varmistaa sen, ettei palvelun resursseihin pääse käsiksi väärennetyllä jwt:llä.

5.2 TypeScript

Valtaosa tämän päivän verkkosivuista hyödyntää toteutuksessaan JavaScript-ohjelmointikieltä, ja kaikki nykyaikaiset selaimet alustasta riippumatta ymmärtävät sitä. JavaScript määrittelee, miten verkkosivut käyttäytyvät. Siksi JavaScriptistä on muodostonut web-kehittäjälle välttämätön työkalu. (Flanagan, 2011, s. 1)

Erityisesti laajojen web-sovellusten kehittämisessä JavaScript aiheuttaa kuitenkin joitakin ongelmia. Koodin hallitseminen ja rakenteen pitäminen siistinä on vaikeaa. Debuggaus on mahdollista vain ajon aikana, mikä hidastaa virheiden löytymistä koodista. Dynaaminen tyypitys edellyttää toisinaan kehittäjältä olioiden sisällön tuntemista ilman, että niistä on olemassa luokkarakennetta. (Nance, 2014, s. 21-22)

Microsoftin kehittämä TypeScript tarjoaa joitakin ratkaisuja näihin ongelmiin. TypeScript on JavaScriptiä laajentava ohjelmointikieli, mikä tarkoittaa sitä, että lähes kaikki syntaktisesti validi JavaScript on myös validia TypeScriptiä mahdollistaen olemassa olevien JavaScript-kirjastojen ja -koodipätkien käytön TypeScript-koodissa. Sen lisäksi TypeScriptissä on joukko ominaisuuksia, jotka helpottavat toimivan ja paremmin jäsennellyn koodin kirjoittamista, sekä parantavat web-sovelluksen skaalautuvuutta. Koska selaimet eivät sellaisenaan ymmärrä TypeScriptiä, TypeScript-koodi on muutettava JavaScript-koodiksi sen sisään rakennetulla kääntäjällä. (Nance, 2014, s. 21-22)

TypeScriptin tarjoamia ominaisuuksia on mm. luokkien määrittely ja sen mahdollistama vahva tyypitys. Nämä ominaisuudet tekevät oliopohjaiseen ohjelmointiin tottuneen

kehittäjän työskentelystä intuitiivisempaa. Koska JavaScript-koodi on yhteensopivaa TypeScriptin kanssa, siirtyessään käyttämään TypeScriptiä JavaScript-kehittäjän ei suinkaan tarvitse ottaa välittömästi käyttöön kaikkia TypeScriptin tarjoamia työkaluja, vaan voi siirtyä niihin vähitellen tai käyttää niitä tarpeen mukaan. (Nance, 2014, s.21-22)

5.3 Angular

Angular on Googlen kehittämä TypeScriptillä toteutettu sovellusalusta. Angular on kokonaan uudelleen kirjoitettu versio AngularJS-kirjastosta. Erotuksena vanhasta AngularJS:stä Angularista käytetään toisinaan nimitystä ”Angular 2”, mutta sen versionumeron jatkuvan päivittymisen vuoksi tästä käytännöstä on pyritty eroon. Angularilla voidaan toteuttaa internet-selaimella toimiva sovellus, joka tarjoaa käyttäjälle näkymiä, jotka pitävät sisällään erilaisia käyttöliittymäelementtejä. Modulaarisen luonteensa vuoksi Angular skaalautuu varsin pitkälle, mikä mahdollistaa hyvin monimutkaistenkin toiminnallisuuksien luomisen selainpohjaisiin sovelluksiin.

Angularin arkkitehtuuri koostuu pääasiassa kolmesta erilaisesta rakennuspalikasta, joita kutsutaan *moduuleiksi, komponenteiksi* ja *serviceiksi*. Nuo kaikki toteutetaan TypeScript-luokkina. (angular.io)

Angular-sovellukseen kuuluu vähintään yksi moduuli, josta käytetän nimitystä root module, juurimoduuli. Moduulin tehtävänä on koostaa yhteen muita Angular-elementtejä, kuten komponentteja ja servicejä, ja mahdollistaa vuorovaikutus niiden välillä. Angular-moduuli voi sisältää myös toisia Angular-moduuleja. (angular.io)

Angular-komponenteissa (component) toteutetaan varsinainen käyttöliittymäsovelluksen logiikka. Oliopohjaisen ohjelmoinnin tapaan komponenteissa on konstruktori, kenttiä ja metodeja. Kentät voivat olla esim. numeerisia arvoja, merkkijonoja, tai TypeScript- tai JavaScript-olioita. Komponenttiin liitetään HTML-pohja (HTML template), jonka avulla loppukäyttäjä pääsee käsiksi komponentin tietoihin. (angular.io)

Välikappaleena Angular-komponenttien ja palvelimella pyörivän backend-sovelluksen välillä toimivat Angularin servicet. Servicet ovat tyypillisesti kokoelmia http-metodeja, jotka ottavat yhteyden palvelimelle ja palauttavat sieltä haettua dataa. Service voidaan injektoida komponenttiin, jolloin servicen palauttama data saadaan asetettua komponentin kenttiin ja sen myötä näyttämään käyttäjälle komponentin HTML-pohjassa. Jos käyttäjän halutaan syöttävän komponenttiin palvelimelle tallennettavia tietoja, myös nuo ajetaan

Angular-servicen kautta esimerkiksi HTTP POST- tai PUT-metodina. Mikäli useammassa komponentissa halutaan hyödyntää samoja HTTP-metodeja, ne kaikki voivat käyttää samaa serviceä, jossa nuo metodit on toteutettu, eikä niissä silloin tarvitse toteuttaa samoja HTTP-kutsuja moneen kertaan. (angular.io)

6 Tuotettu sovellus

Koodittaren projektinhallintasovellusta on kehitetty normaalin kehityskaaren mukaisesti pääosin iteratiivisin menetelmin. Sovelluksen kehityksen aikana ilmenneet haasteet sekä osittain teknologiavalinnat ovat edellyttäneet kehityskaaren aikaisempiin vaiheisiin palaamista, mikä on yleistä ketterissä ohjelmistokehitysprojekteissa. Esimerkiksi Entity Frameworkin code-first-lähestymistavan johdosta suunnitteludokumentteihin kuuluva tietokantakaavio on luotu jo toteutettujen C#-luokkien perusteella.

6.1 Määrittely

Tämän opinnäytetyön puitteissa toteutetaan ns. minimum viable product Koodittaren projektinhallintasovelluksesta. MVP-versioon toteutetaan joukko käyttötapauksia, jotka mahdollistavat sovelluksen peruskäytön. Sovelluksella voi olla viisi eri tyyppistä käyttäjäryhmää, joilla on eri käyttöoikeuksia: kirjautumaton käyttäjä, kirjautunut käyttäjä, projektin käyttäjä, projektin hallinoija ja palvelun ylläpitäjä.

Käyttäjän kuuluminen yhteen käyttäjäryhmään ei sulje pois tämän kuulumista toiseen käyttäjäryhmään. Käyttäjäryhmien eri käyttöoikeudet etenevät hierarkisesti siten, että esimerkiksi projektin hallinnoijan käyttöoikeuksiin kuuluvat kaikki samaan projektiin liittyvän projektin käyttäjän oikeudet, ja palvelun ylläpitäjän käyttöoikeuksiin kuuluvat kaikkien projekteiden projektin hallinnoijan käyttöoikeudet. Käyttäjäryhmät on toteutettu sovelluksessa rooleina (tietokannan taulu AspNetRoles). Taulukossa 1 on esitelty toteutetut käyttötapaukset käyttäjätarinoina.

Toimija	**Tarina**
palvelun ylläpitäjä	Palvelun ylläpitäjänä haluan luoda palveluun uuden käyttäjän
kirjautumaton käyttäjä	Kirjautumattomana käyttäjänä haluan kirjautua palveluun
kirjautunut käyttäjä	Kirjautuneena käyttäjänä haluan luoda palvelussa uuden projektin
kirjautunut käyttäjä	Kirjautuneena käyttäjänä haluan nähdä listan projekteista, joihin minulla on käyttö- tai hallinnointioikeudet
projektin hallinnoija	Projektin hallinoijana haluan tehdä toisesta projetkin käyttäjästä projektin hallinnoijan

projektin käyttäjä	Projektin käyttäjänä haluan antaa projektin käyttöoikeudet toiselle palvelun käyttäjälle
projektin käyttäjä	Projektin käyttäjänä haluan luoda uuden työvaiheen projektiin
projektin käyttäjä	Projektin käyttäjänä haluan luoda uuden tehtävän projektiin
projektin käyttäjä	Projektin käyttäjänä haluan lisätä toisen projektin käyttäjän johonkin työtehtävään
projektin käyttäjä	Projektin käyttäjänä haluan kirjata ylös johonkin tehtävään käyttämääni aikaa
projektin käyttäjä	Projektin käyttäjänä haluan siirtää tehtävän yhdestä työvaiheesta seuraavaaan
projektin käyttäjä	Projektin käyttäjänä haluan arkistoida tehtävän, joka on saatu valmiiksi

Taulukko 1 Käyttäjätarinat

Käyttötapauksia on havainnollistettu käyttötapauskaaviossa (liite 3). Kaaviossa on yhdistetty kirjautuneen käyttäjän ja projektin käyttäjän käyttötapaukset samaan toimijaan.

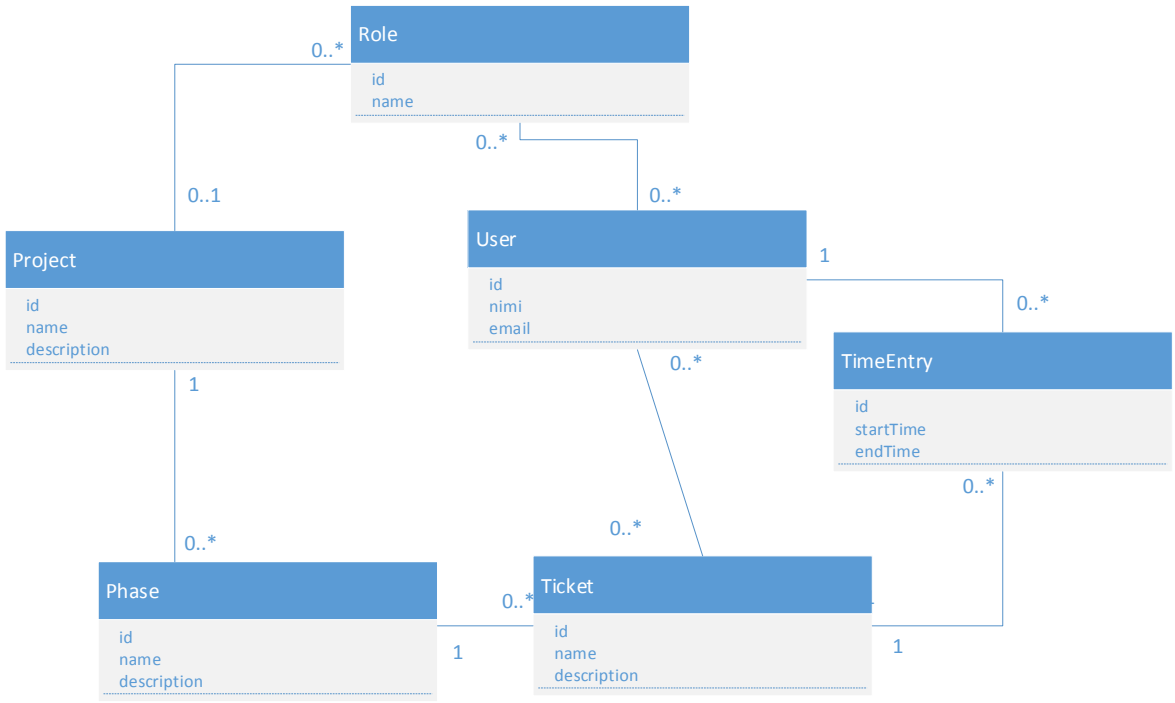
6.2 Suunnittelu

Sovelluksen suunnitteluvaiheeseen kuuluu luokkamallinnuksen, tietokantamallin ja käyttöliittymäsuunnitelmien laatiminen. Luokkamalli ja käyttöliittymäsuunnitelma on laadittu normaalissa kehityskaaren mukaisessa järjestyksessä ennen toteutuksen aloittamista. Tietokantamalli puolestaan puolestaan perustuu Entity Frameworkin luomaan tietokantaan, joka on johdettu jo toteutetuista C#-luokista.

6.2.1 Luokkarakenne

Sovelluksen luokkarakenne (Kuva 8) perustuu projekteihin ja niihin liittyviin eri elementteihin. Sovellukseen tarvitaan myös käyttäjähallintaan liittyviä luokkia, sillä projektien näkyvyyttä halutaan rajoittaa käyttäjän perusteella. Sovelluksen pääluokkana voidaan pitää projektia (Project), johon liittyy työvaiheita (Phase). Työvaiheita ei voi luoda ilman, että ne liittyvät johonkin projektiin. Työvaiheessa on puolestaan tikettejä (Ticket), joiden on pakko olla merkittynä johonkin työvaiheeseen. Projektiin käytetty työaika tallennetaan aikamerkintöihin (TimeEntry), jotka liittyvät aina johonkin tikettiin.

Projektilla voi olla käyttäjiä (User), jotka yhdistetään projektiin roolin (Role) kautta. Roolista riippuen käyttäjällä voi olla erilaisia oikeuksia projektiin. Roolin ei kuitenkaan ole pakko liittyä mihinkään projektiin, koska esimerkiksi palvelun ylläpitäjällä (admin) on pääsy kaikkiin projekteihin ilman, että sitä tarvitsee erikseen spesifioida. Käyttäjä voidaan liittää johonkin tikettiin, ja samaan tikettiin voidaan liittää useita käyttäjiä, joten luokkien käyttäjä ja tiketti välillä vallitsee monen suhde moneen -yhteys. Aikamerkinnät sen sijaan ovat käyttäjäkohtaisia, joten jokaiseen aikamerkintään liittyy aina yksi ja vain yksi käyttäjä.



Kuva 8 Luokkakaavio

6.2.2 Tietokanta

Projektin tietokantamalli on laadittu .NETin Entity Framework -työkalulla ”Code first” -lähestymistapaa hyödyntäen. Tietokanta on luotu suoraan C#-koodista DbContext-luokasta käsin, minkä johdosta tietokantamalli vastaa sovelluksen luokkamallia täysin yhtenevästi.

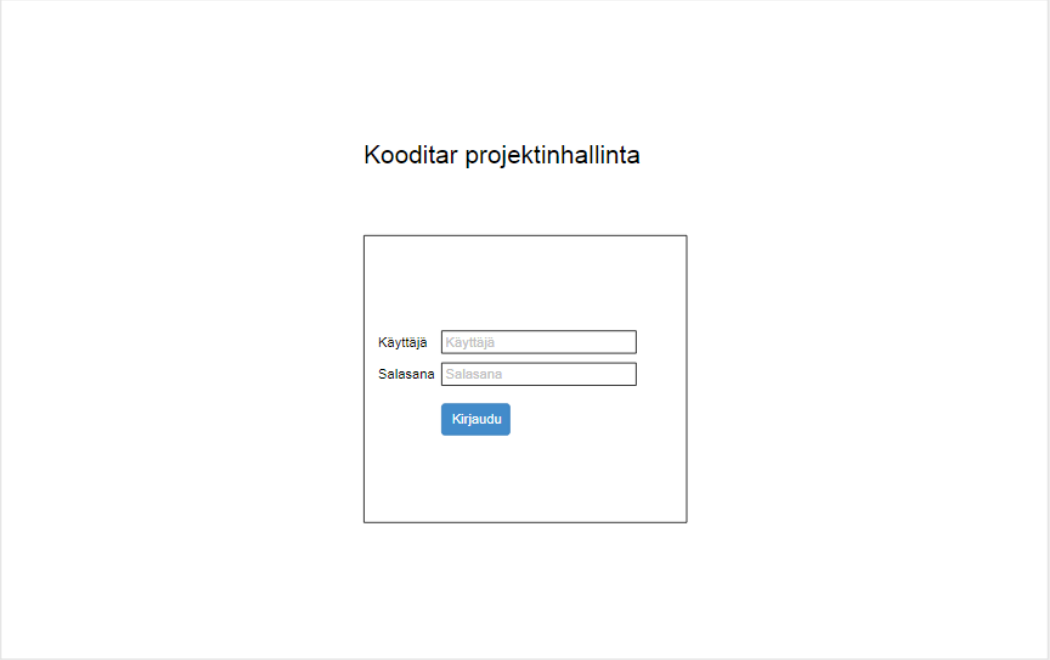
Tietokantamalli (liite 1) vastaa pääpiirteissään suunniteltua luokkarakennetta. (ks. edellinen luku). Monen suhde yhteen -yhteyksissä täytyy luoda tietokantatauluihin viiteavaimet. Taulujen User ja Ticket sekä taulujen User ja Role välillä vallitsee monen suhde moneen -yhteydet, joten näitä yhteyksiä varten tarvitaan tietokantaan välitaulut UserTicket ja AspNetUserRoles. Tietokantataulut ”AspNetUsers”, ”AspNetRoles” ja

”AspNetUserRoles” ovat .NETin Identity-kirjaston toteuttamia tauluja, jotka kuvastavat User- ja Role-luokkia, sekä niiden välistä välitaulua. Luokkamallissa ei ole suoraa yhteyttä luokkien Project ja Ticket välillä, eikä myöskään luokkien Project ja TimeEntry välillä. Tällaisten yhteyksien luominen tietokantaan on kuitenkin havaittu hyödylliseksi, sillä se esimerkiksi helpottaa projektiin käytetyn kokonaisajan laskemista ilman, että tietoa täytyy hakea useiden eri viiteavaimien ja taulujen kautta. Tietokantataulujen kentät, pääavaimet ja viiteavaimet on selitetty auki tietokannan tietohakemistokuvauksessa (liite 2).

6.2.3 Käyttöliittymäsuunnitelma

Sovelluksen käyttöliittymä laaditaan web-sivustona TypeScript-kielellä Angular-viitekehystä hyödyntäen. HTML-sivut ja tyylitiedostot tehdään osana Angular-komponenttejä. Tässä luvussa esittellään vaihe vaiheelta käyttöliittymän rautalankamalli sovelluksen peruskäyttäjän osalta. Mallin ei ole tarkoitus olla ulkoasultaan yhdenmukainen toteutuneen käyttöliittymän kanssa, vaan suuntaa-antava ohje käyttöliittymän kehittäjille siitä, minkälaisia toiminnallisuuksia siinä tulee olla. Käyttöliittymäsuunnitelmassa on huomioitu projektin MVP-luonne, eli se kattaa vain sovelluksen toiminnallisuuden kannalta kaikkein oleellisimmat käyttötapaukset. Käyttöliittymää kehitettäessä tullaan myös huomioimaan käyttäjäkokemukseen liittyvät parannusehdotukset, joten toteutettu versio voi niiltä osin poiketa tässä suunnittelusta. Lopullinen versio käyttöliittymästä toteutetaan mahdollisten jatkokehitysvaiheiden aikana. Käyttöliittymäsuunnitelma on luoto Moqups-työkalun ilmaisversiolla.

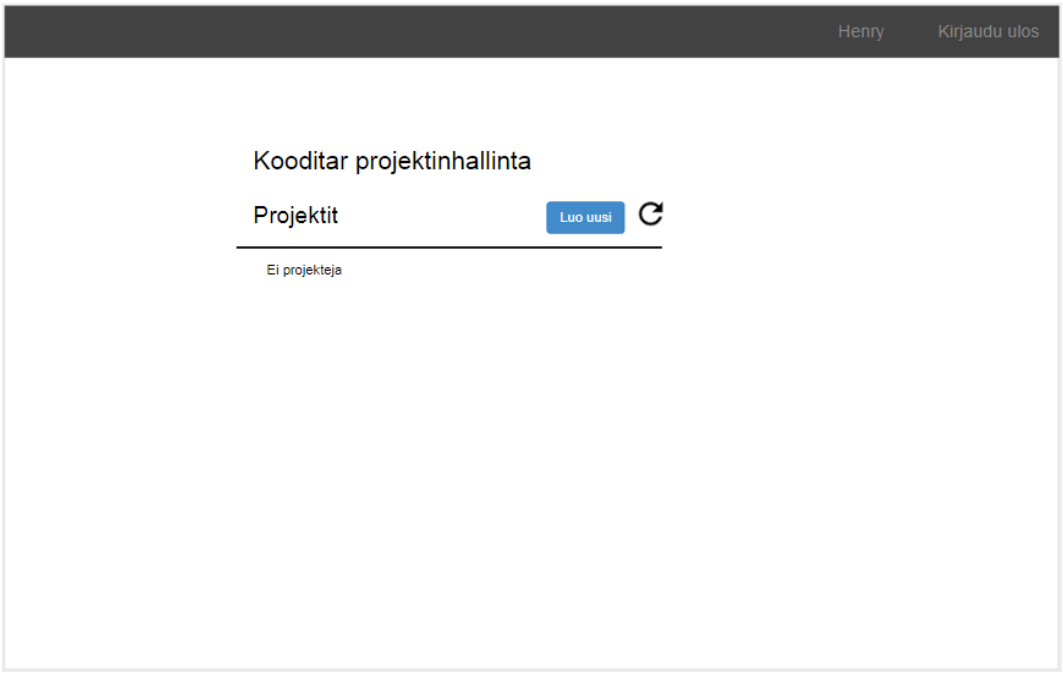
Alkutilanteessa palvelun ylläpitäjä on luonut käyttäjälle käyttäjätunnuksen ja salasanan, joilla hän voi kirjautua sisään (kuva 9). Rekisteröitymismahdollisuutta ei haluta ainakaan aluksi toteuttaa, koska palvelun ensimmäistä versiota ei haluta yleiseen käyttöön.



Kuva 9 Kirjautumisnäkymä

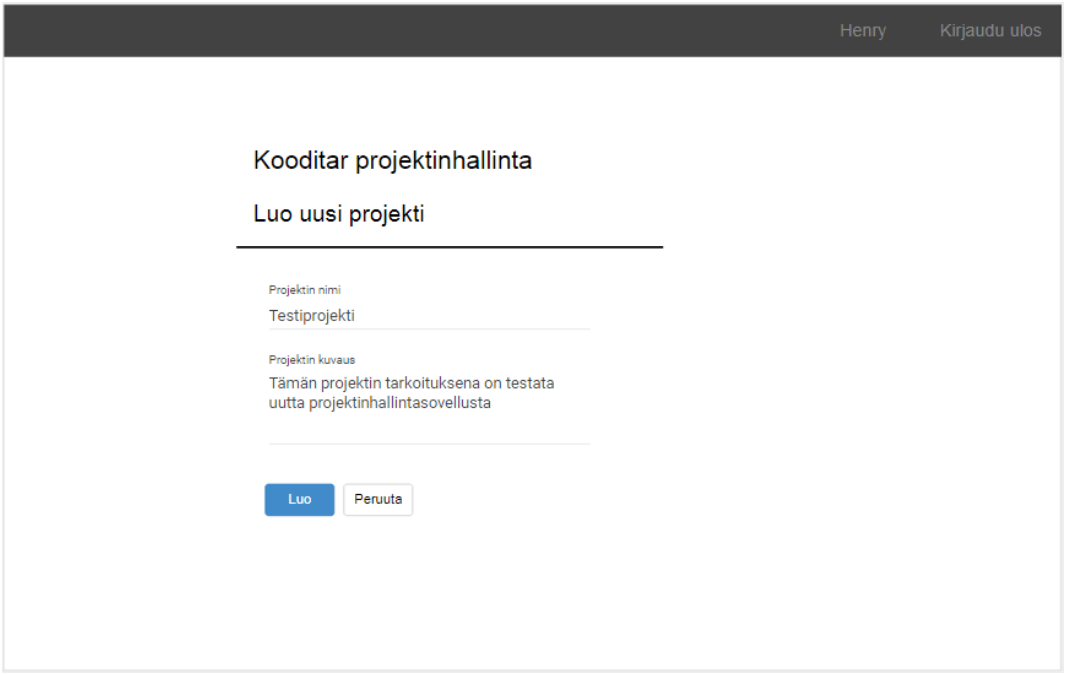
Kirjauduttuaan palveluun käyttäjälle ilmestyy näkymä, jossa on listattuna projektit, joihin hänellä on käyttöoikeudet. Jos käyttäjällä ei ole pääsyä yhteenkään projektiin, näkymä antaa ”Ei projekteja” -ilmoituksen (kuva 10). Näkymässä on myös painike uuden projektin luomiselle, ja sen vieressä painike, jolla käyttäjä voi päivittää näkymän. Jos käyttäjälle on annettu käyttöoikeudet johonkin projekteihin sillä aikaa, kun hän on selannut sivua, nuo projektit tulevat hänelle näkyviin, kun näkymä päivittyy.

Sivun oikeasta yläkulmasta käyttäjä voi kirjautua ulos, tai painamalla oman käyttäjänimensä kohdalta muokata omia tietojaan, kuten vaihtaa sähköpostiosoitettaan tai salasanaa.



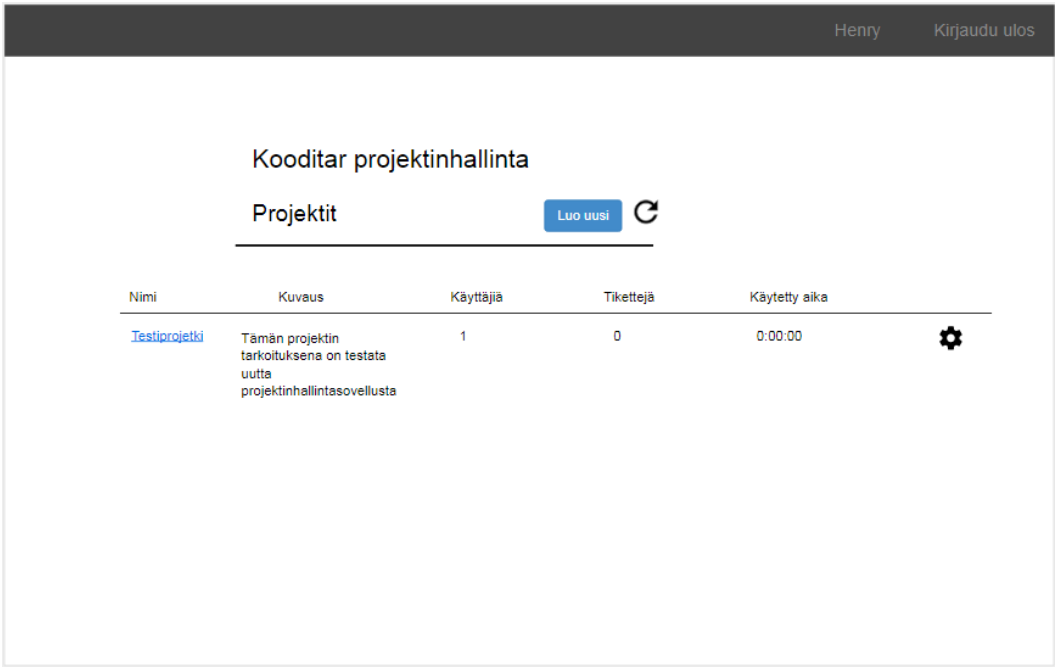
Kuva 10 Tyhjä projektinäkymä

Kun käyttäjä painaa Luo uusi -painiketta, hänelle aukeaa näkymä, jossa hän voi luoda uuden projektin. Käyttäjä voi antaa projektille nimen ja kuvauksen. Luotuaan projektin käyttäjä palaa sivulle, jossa projektit on listattuna. Painamalla Peruuta-painiketta, käyttäjä palaa samalle sivulle ilman, että järjestelmään luodaan uutta projektia. (Kuva 11)



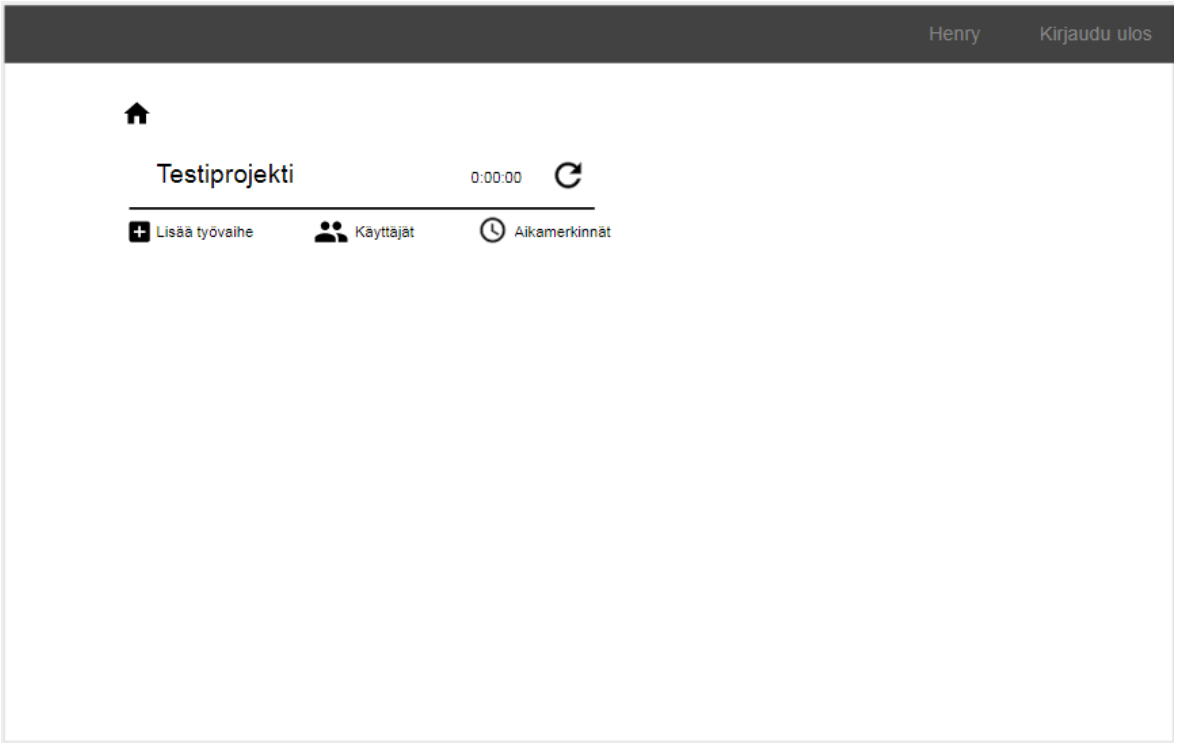
Kuva 11 Sivu, jossa luodaan uusi projekti

Palattuaan projektien listausnäkymään uuden projektin luomisen jälkeen luotu projekti ilmestyy näkymään. Käyttäjä saa automaattisesti oikeudet luomaansa projektiin. Projektista on listattuna sen nimi, kuvaus, käyttäjien määrä, luotujen tikettien määrä ja siihen käytetty aika. Hammasratas-painikkeella käyttäjä voi muokata projektin nimeä tai kuvausta. Projektin nimi näytetään linkkinä. Napsauttamalla tuosta linkistä käyttäjä siirtyy projektin yksityiskohtaisempaan näkymään. (Kuva 12)



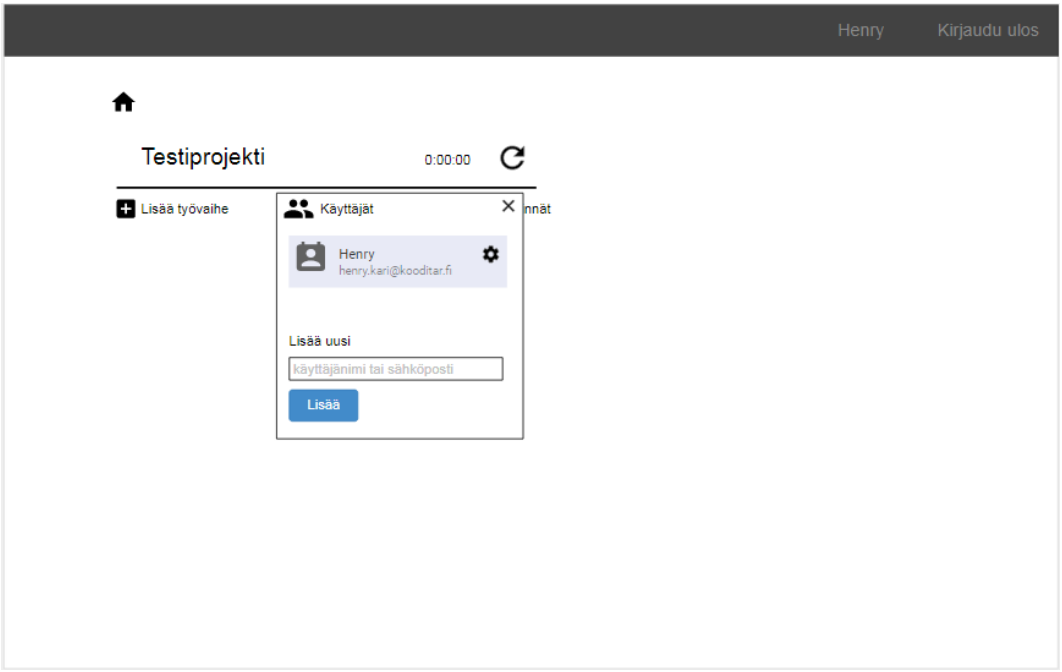
Kuva 12 Projektit-näkymä, kun uusi projekti on luotu

Projektin yksityiskohtaisessa näkymässä käyttäjälle näkyy projektiin käytetty aika, työvaiheet, tiketit, käyttäjienhallintavalikko sekä linkki aikamerkintöjen hallintasivulle. Koska projekti on vasta luotu, siinä ei vielä ole työvaiheita tai tikettejä. Lisää työvaihe -painikkeesta käyttäjä voi luoda uuden työvaiheen. Tikettejä ei voida luoda, ennen kuin projektissa on vähintään yksi työvaihe. Päivitysnappulasta käyttäjä näkee projektin päivitttyneen tilan, jos joku toinen sen käyttäjistä on esimerkiksi lisännyt sinne työvaiheita tai tikettejä. Koti-painikkeesta käyttäjä pääsee takaisin projektin listausnäkymään. (Kuva 13)



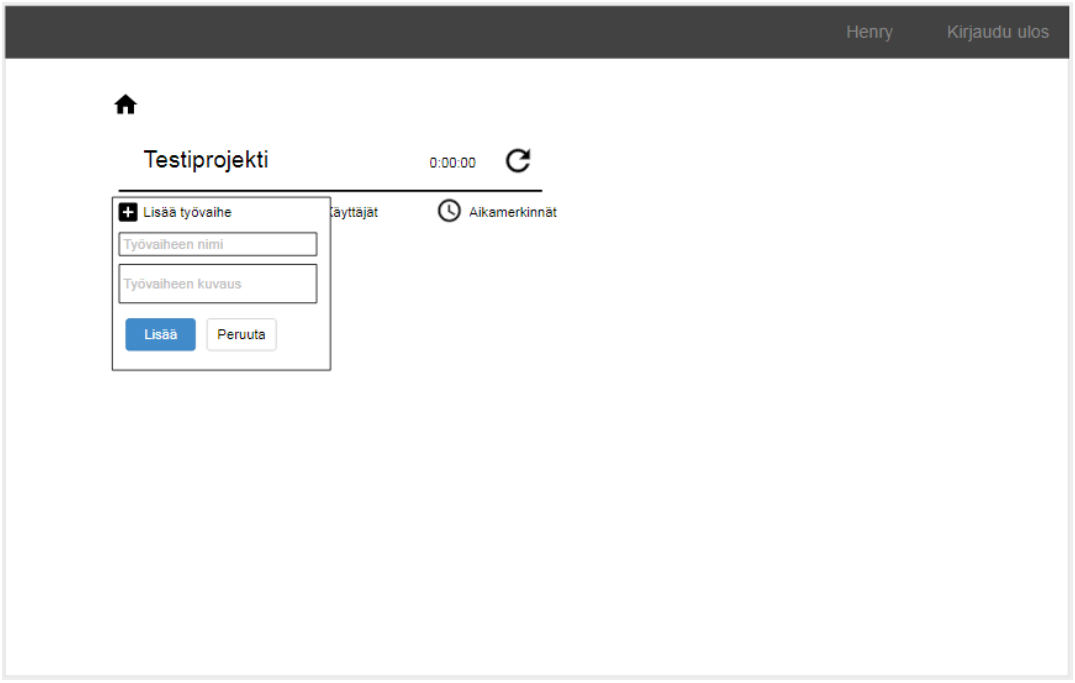
Kuva 13 Näkymä luodusta projektista

Käyttäjät-valikosta käyttäjä voi lisätä itselleen tiimitovereita projektiin. Käyttäjä syöttää kenttään lisättävän käyttäjän käyttäjänimen tai sähköpostiosoitteen. Lisättävän käyttäjän tulee löytyä järjestelmästä. (Kuva 14)



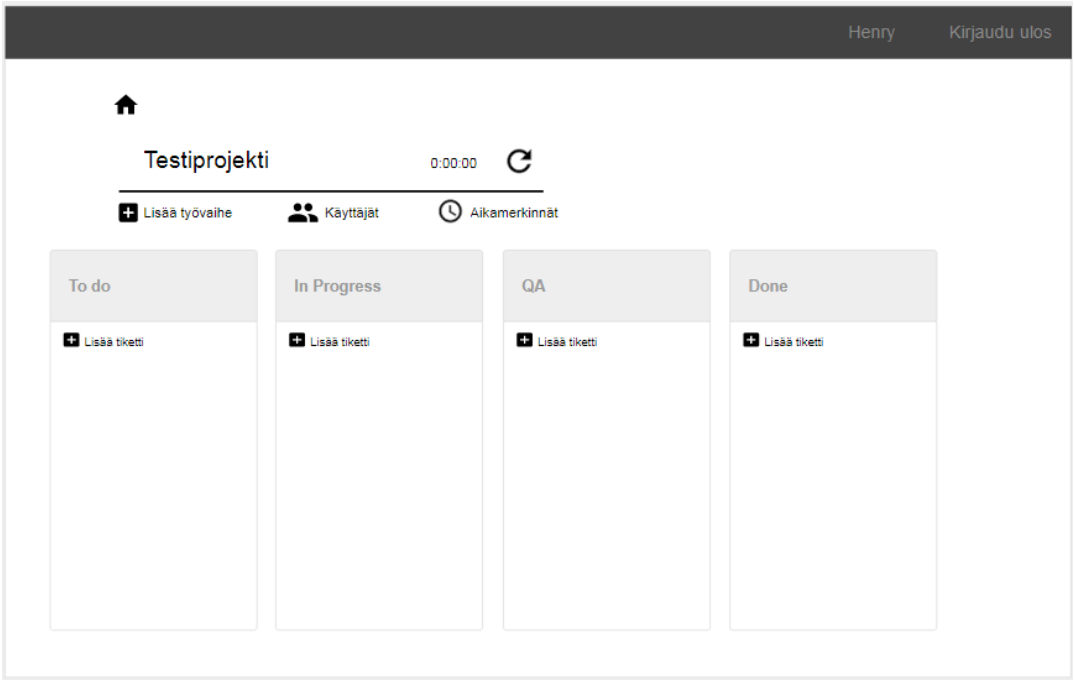
Kuva 14 Projektin käyttäjävalikko

Painamalla Lisää työvaihe -painiketta, käyttäjälle avautuu pieni modaali-ikkuna. Siinä on kaksi kenttää, johon hän voi syöttää uuden työvaiheen nimen ja kuvauksen. (Kuva 15)



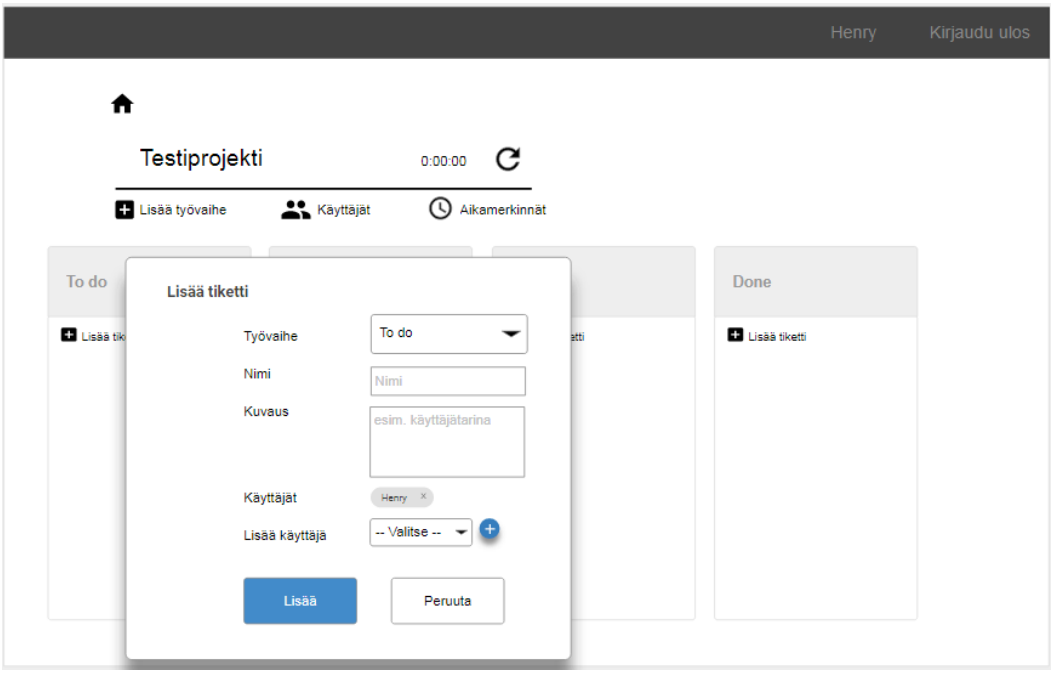
Kuva 15 Uuden työvaiheen lisääminen

Luodut työvaiheet muodostavat uimaratoja muistuttavan sommitelman näkymään. Tämä voi kuvastaa esimerkiksi kanban-taulua. Kuhunkin työvaiheeseen voi lisätä tikettejä. (Kuva 16)



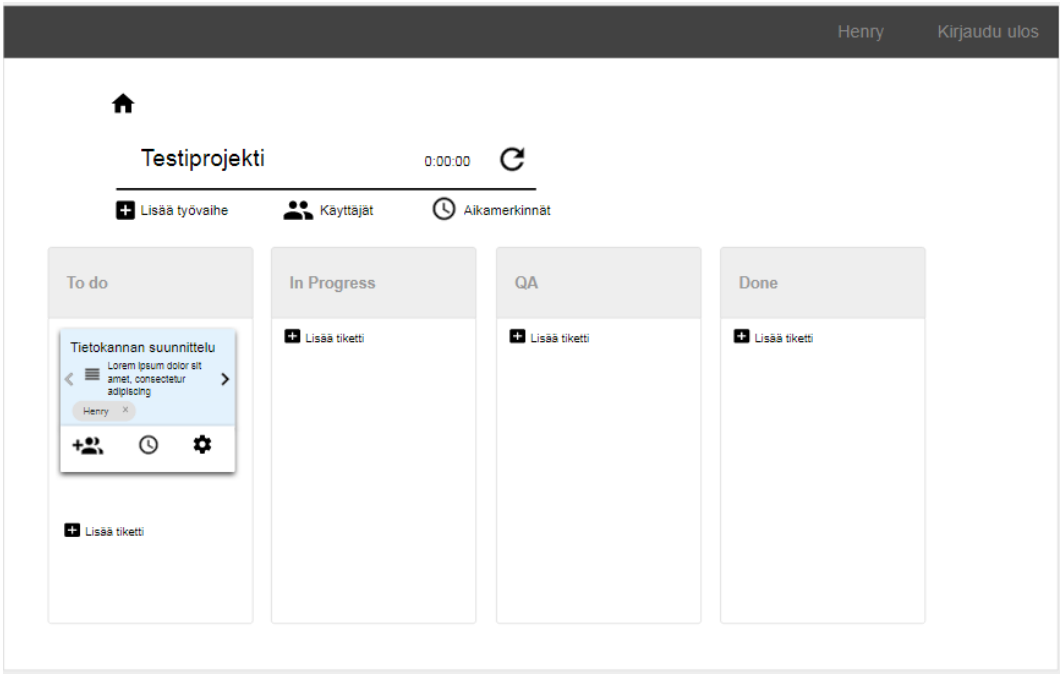
Kuva 16 Luotuja työvaiheita

Lisää tiketti -painikkeesta käyttäjälle avautuu modaali, jolla tiketin voi luoda. Käyttäjä antaa tiketin nimen ja kuvauksen, sekä valitsee käyttäjät, jotka tiketissä ovat mukana, oletuksena vain käyttäjä itse. Jos tiketti halutaankin luoda johonkin toiseen työvaiheeseen, niin sitä voi tässä vaiheessa vielä vaihtaa työvaiheen pudotusvalikosta. (Kuva 17)



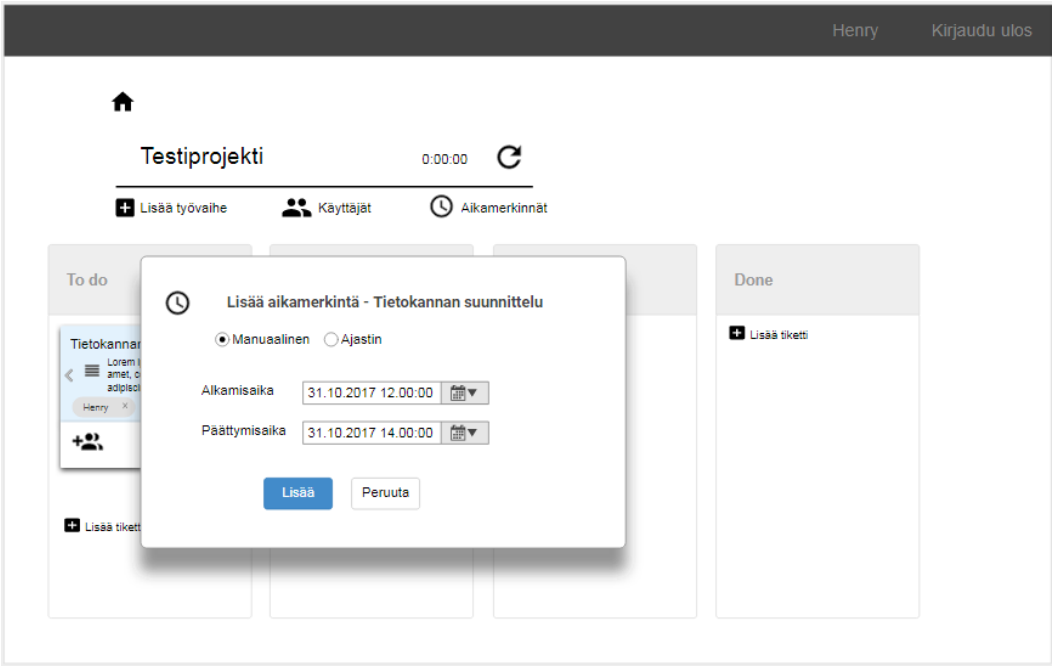
Kuva 17 Tiketin luominen

Kun tiketti on luotu, se ilmestyy valittuun työvaiheeseen. Tiketissä on näkyvillä sen nimi, kuvaus ja käyttäjät. Tiketin vasemmalla ja oikealla reunalla on nuolipainikkeet, joista sitä voi siirtää toiseen työvaiheeseen. Tiketin alareunassa on valikko, josta voi lisätä tikettiin käyttäjiä, merkitä tikettiin käytettyä aikaa tai muokata tiketin nimeä tai kuvausta. (Kuva 18)



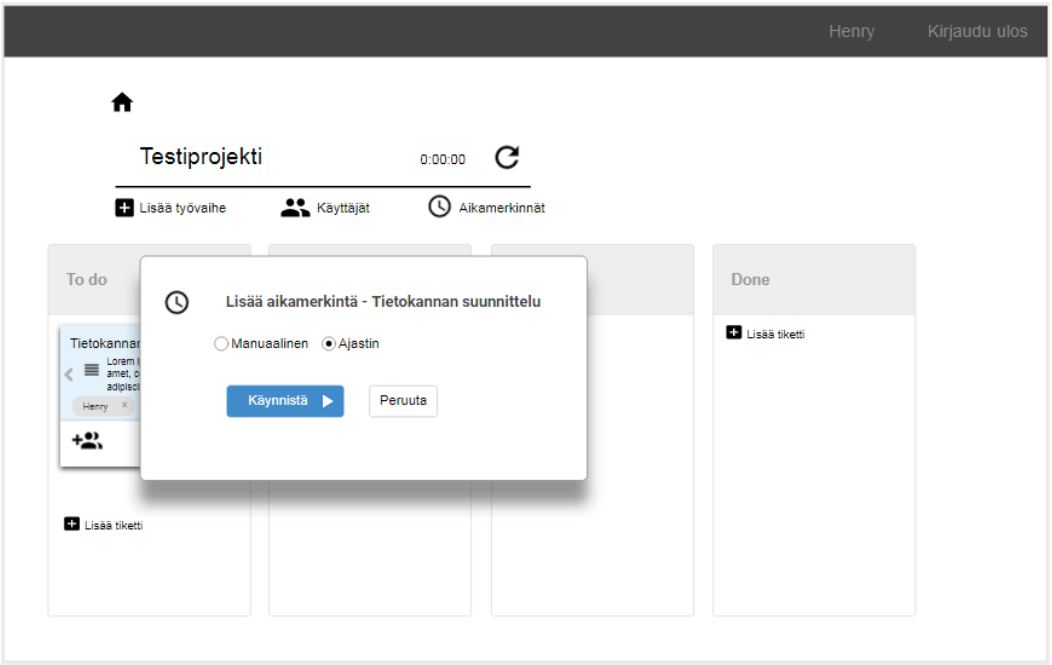
Kuva 18 Luotu tiketti

Painamalla tiketin kelloikonia käyttäjälle avautuu modaali, jolla hän voi lisätä aikamerkinnän järjestelmään. Käyttäjä voi valita aikamerkinnän lisäämisen manuaalisesti tai ajastimella. Manuaalisessa vaihtoehdossa käyttäjä valitsee aikamerkinnän alkamis- ja päättymisajankohdat ja syöttää ne järjestelmnään. (Kuva 19)



Kuva 19 Aikamerkinnän lisääminen

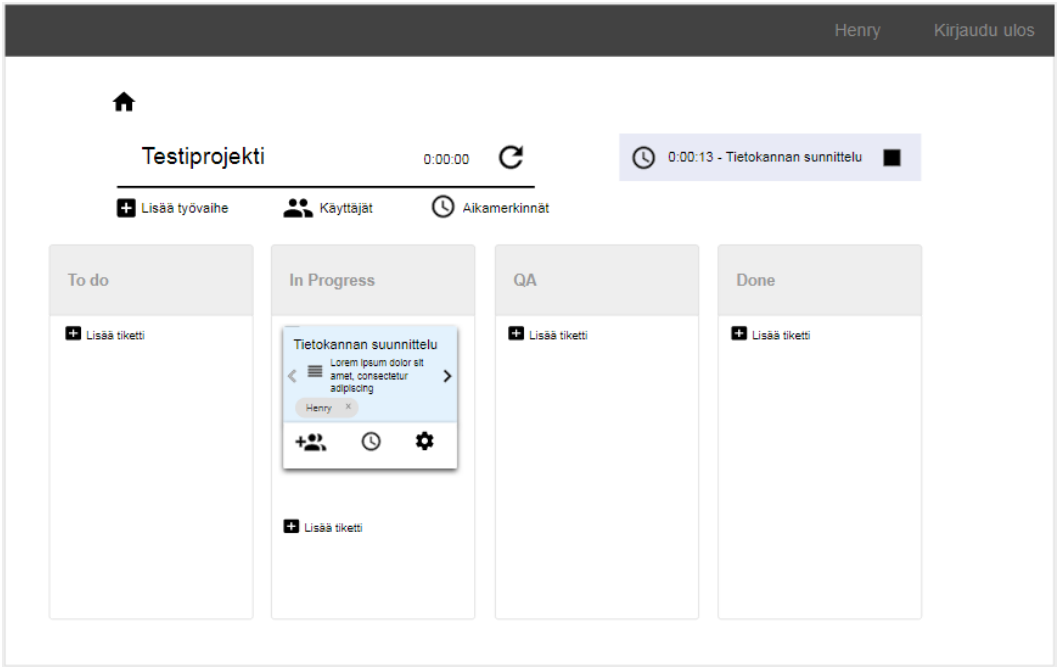
Jos käyttäjä valitsee aikamerkinnän lisäämisen ajastimella, alkamis- ja päättymisajankohtien valinnat piiloutuvat käyttäjältä, ja näkymään ilmestyy painike, jolla ajastimen voi laittaa käyntiin. (Kuva 20)



Kuva 20 Aikamerkinnän lisääminen ajastimella

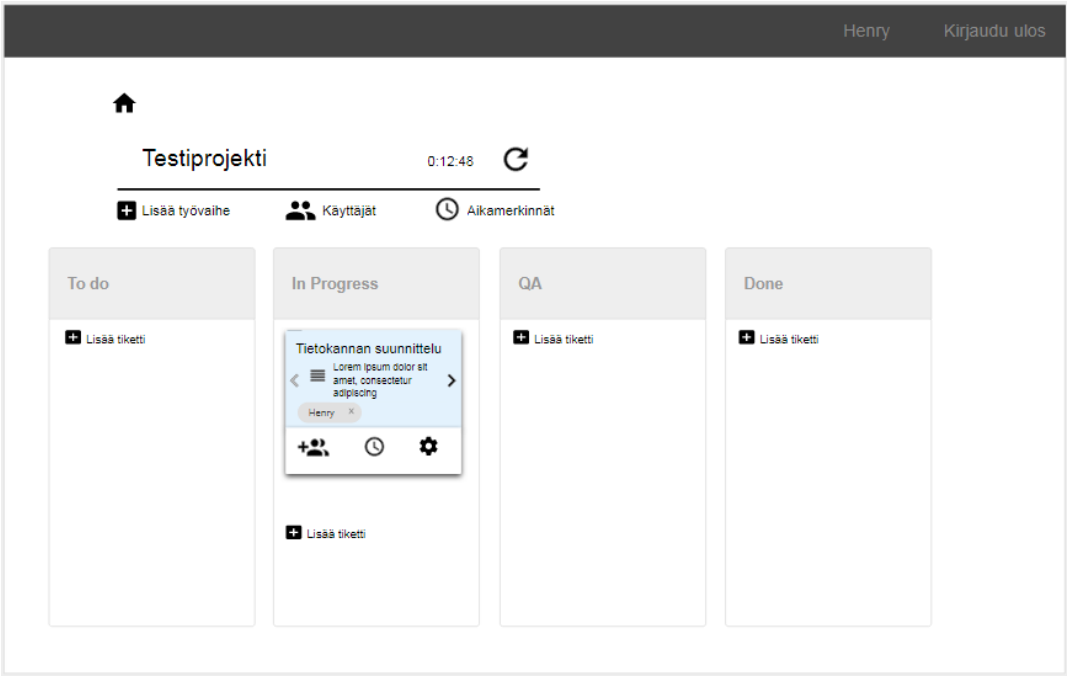
Käyttäjän painettua ajastimen käytiin, sivun yläreunaan ilmestyy pieni ilmoitus, josta näkyy, että ajastin on käynnissä. Ajastimen voi pysäyttää ilmoitukseen liitetystä stop-

painikkeesta. Jos käyttäjä sulkee sovelluksen ajastimen vielä ollessa käynnissä, se ei pysähdy automaattisesti, vaan käyttäjän on avattava sovellus uudestaan voidakseen pysäyttää ajastimen. Mikäli sovellusta käytetään kanban- tai scrum-tauluna, tiketti on syytä myös siirtää ”In Progress” -vaiheeseen. (Kuva 21)



Kuva 21 Ajastin käynnissä

Kun käyttäjä on pysäyttänyt ajastimen tai lisännyt aikamerkinnän manuaalisesti, aikamerkintään käytetty lisätään projektiin käytettyyn kokonaisaikaan (Kuva 22). Aikaan lasketaan yhteen projektin kaikkien käyttäjien syöttämät aikamerkinnät. Jos käyttäjä syöttää virheellisen aikamerkinnän, hän pääsee poistamaan sen tai korjaamaan sitä aikamerkinnät-valikosta.



Kuva 22 Aikamerkintä syötetty

6.3 Toteutus

Sovelluksesta toteutettiin REST-rajapinta .NET Core -projektina. Sovelluksen käyttöliittymän runko rakennettiin Angular-sovellusalustalla. Käyttöliittymän lopullinen toteutus tehdään myöhempänä ajankohtana.

6.3.1 Back-end

Sovelluksen backend-lähdekoodi on toteutettu .NET Core -projektina C#-kielellä Visual Studio -ohjelmointiympäristössä. .NET-projekti pitää sisällään luokkamallin sekä REST-rajapinnan. Tietokanta on generoitu suoraan C#-luokista. Kuvassa 23 on Project-luokan toteus C#:lla. Vastaavat C#-luokat on tehty kaikista sovelluksessa käytettävistä luokista (ks. luku 6.2.1) sekä niiden välille tarvittavista välitauluista (ks. luku 6.2.2). Nämä luokat vastaavat MVC-mallin ”Model”-osaa.

```
public class Project
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public List<Phase> Phases { get; set; }
    public List<Ticket> Tickets { get; set; }
    public List<TimeEntry> TimeEntries { get; set; }
}
```

Kuva 23 Project-luokka Tregl-sovelluksesta

Sovelluksen tietokantakonfiguraatio on toteutettu TreglContext-nimisenä luokkana (kuva 24), joka periytyy IdentityDbContext-luokasta. IdentityDbContext puolestaan on Entity Framework-kirjaston toteuttama DbContext-luokan aliluokka (ks. luku 5.1.2). TreglContext-luokassa määritellyissä DbSet-tyyppisissä kentissä tuodaan ilmi, mistä kaikista sovelluksen luokista generoidaan taulut tietokantaan. Toisin kuin standardi .NETin Entity Framework, Entity Framework Core ei osaa generoida tietokannan välitauluja automaattisesti, vaan luokan OnModelCreating-metodissa täytyy kertoa, että luokkien User ja Ticket välillä on monen suhde moneen -yhteys ja niille täytyy luoda välitaulu UserTicket.

```
public class TreglContext : IdentityDbContext<ApplicationUser, ApplicationRole, string>
{
    public TreglContext(DbContextOptions<TreglContext> options) : base(options)
    {
    }

    public DbSet<Project> Projects { get; set; }
    public DbSet<Phase> Phases { get; set; }
    public DbSet<Ticket> Tickets { get; set; }
    public DbSet<TimeEntry> TimeEntries { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<UserTicket>()
            .HasOne(ut => ut.User)
            .WithMany(u => u.UserTickets);

        modelBuilder.Entity<UserTicket>()
            .HasOne(ut => ut.Ticket)
            .WithMany(t => t.UserTickets);

    }
}
```

Kuva 24 TreglContext-luokan toteutus

Sovelluksen Controller-luokissa (esim. ProjectsController) käydään käsiksi tietokantadataan TreglContext-olion avulla, joka injektoidaan luokkaan sen konstruktorissa (kuva 25). Controllerin HTTP-metodit palauttavat dataa JSON-muodossa, joka voidaan hakea käyttöliittymään käyttöliittymäsovelluksen suorittamilla HTTP-kutsuilla.

```
public class ProjectsController : Controller
{
  private readonly UserManager<ApplicationUser> _userManager;
  private TreglContext _context;

  public ProjectsController(TreglContext context,
    UserManager<ApplicationUser> userManager,
    RoleManager<ApplicationRole> roleManager)
  {
    _context = context;
    _userManager = userManager;
  }

  // luokan muu sisältö piilotettu näkyvistä
  ...
}
```

Kuva 25 ProjectsController-luokka ja sen konstruktori

Esimerkiksi ProjectsControllerin Get-metodi (kuva 26) palauttaa JSON-muodossa listan projekteista, joihin käyttäjällä on oikeudet. Käyttäjän autorisointi sekä tokenin palauttaminen (JSON Web Token, ks. luku 5.1.4) on toteutettu JwtHandler-nimisessä luokassa, johon metodi viittaa. Tokeniin on tallennettu Claim-tyyppisiin olioihin mm. käyttäjän oikeudet eri projekteihin. Metodin yhteyteen määritellyssä Route-attribuutissa kerrotaan, minkä url-polun kautta metodiin pääsee käsiksi. Jos sovellus olisi julkaistu esimerkiksi osoitteessa www.kooditar.fi/tregl, ProjectsControllerin Get-metodia kutsuttaisiin silloin osoitteesta www.kooditar.fi/tregl/api/projects. Koska projekteja ei näytetä kirjautumattomille käyttäjille, HTTP-kutsussa on oltava header-attribuuttina edellämainittu JSON Web Token, jotta metodi palauttaa jotain dataa käyttäjälle.

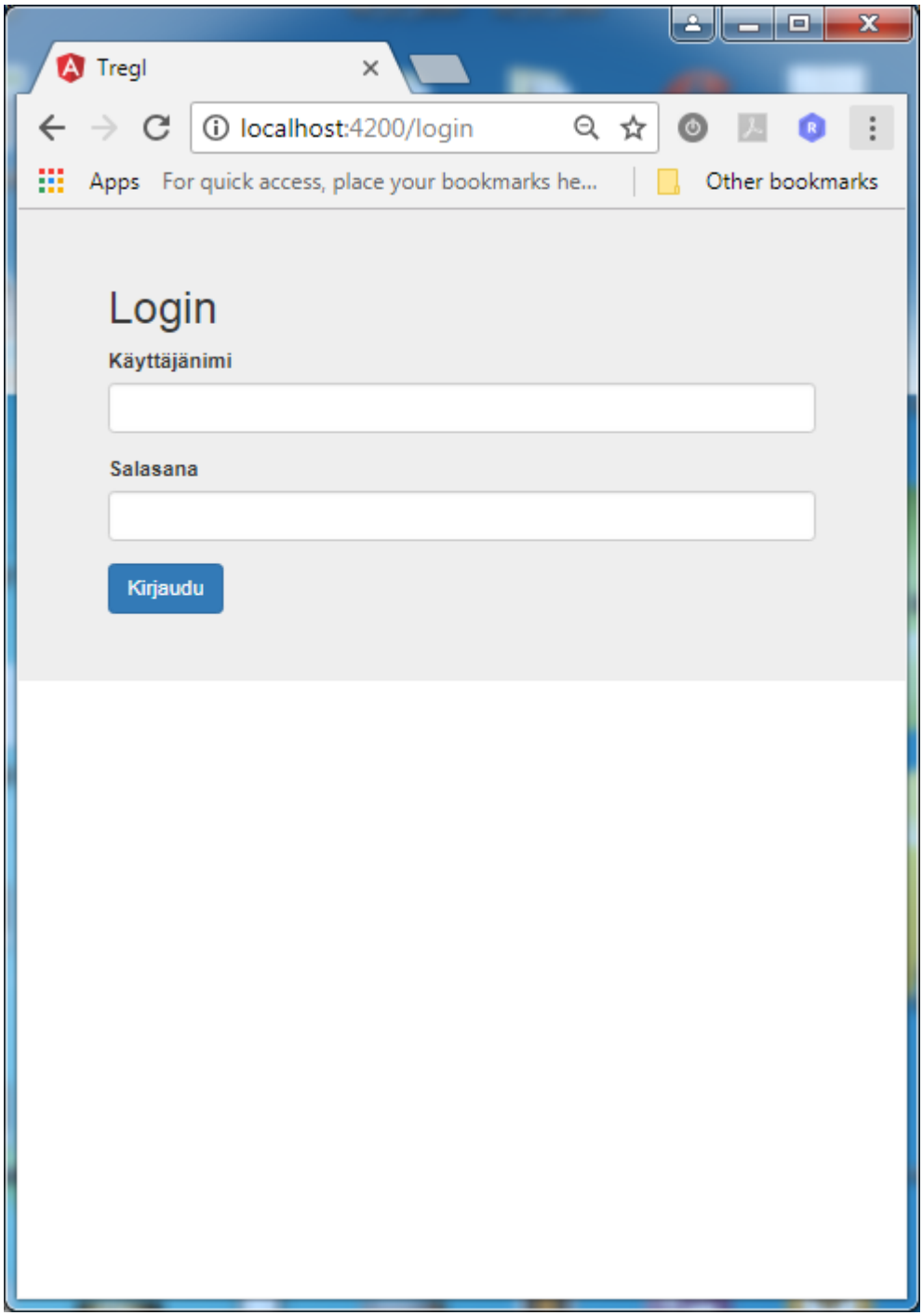
```
[Route("api/projects")]
[HttpGet]
public ActionResult Get()
{
  if (JwtHandler.AuthorizeAdmin(Request.Headers))
  {
    var projects = _context.Projects.ToList();
    return new OkObjectResult(projects.Select(p => new ProjetDTO(p)));
  }

  if (JwtHandler.Authorize(Request.Headers))
  {
    var projects = new List<Project>();
    var token = JwtHandler.GetToken(Request.Headers);
    foreach (var claim in token.Claims)
    {
      if (claim.Type != "projectuser" && claim.Type != "projectadmin") continue;
      var projectId = claim.Value;
      var project = _context.Projects.FirstOrDefault(p => p.Id.ToString() == projectId);
      projects.Add(project);
    }
    return new OkObjectResult(projects.Select(p => new ProjetDTO(p)));
  }

  return Unauthorized();
}
```

Kuva 26 ProjectsController-luokan Get-metodi

Sovellukseen on toteutettu kaikkien luvussa 6.1.1 esitettyjen käyttötapauksien vaatimat Controllerit ja HTTP-metodit. Muutamia sovelluksen REST-reittejä on esitelty liitteessä 4, jossa tuodaan ilmi, minkälaisia metodeja rajapinnassa on, minkälaista dataa HTTP-pyyntöjen on pidettävä sisällään ja minkälaista dataa palvelimelta saa vastauksena. Liitteessä 5 on puolestaan esitelty sovelluksen koko luokkakaavio sisältäen myös Controller- ja DTO-luokat. DTO-luokista käytetään tässä kaaviossa nimitystä ”Contract”, (esim. ProjectContract) vaihtoehtoisen nimeämiskäytännön mukaisesti, mutta näiden luokkien nimet on jälkikäteen muutettu muotoon ”DTO” (esim. ProjectDTO).

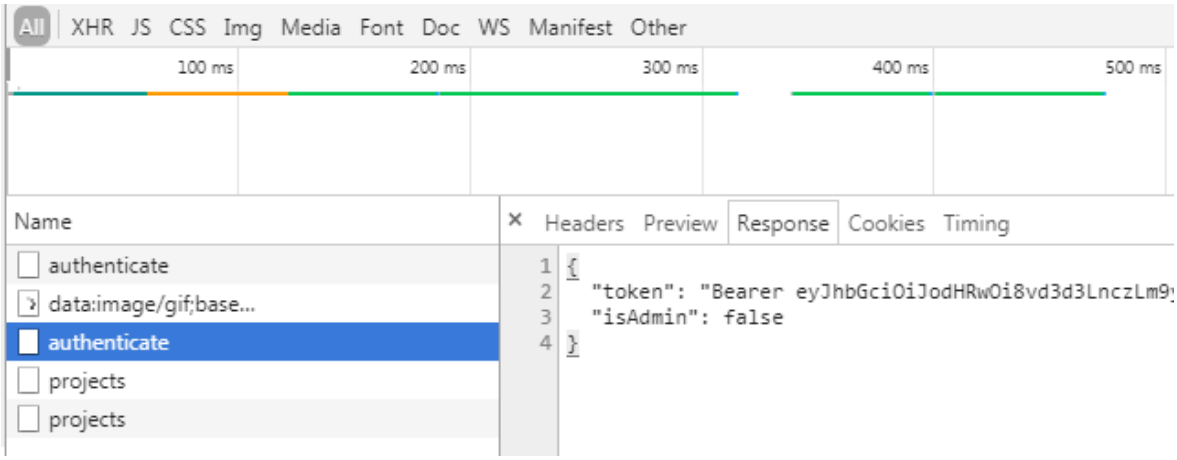
6.3.2 Käyttöliittymä (front-end)

Käyttöliittymän runko on toteutettu www-sivustona Angular-sovelluskehyksellä, joka hyödyntää HTML, CSS ja JavaScript/TypeScript-kieliä. Front-end-projekti on eriytetty kokonaan back-endistä, eikä sitä ajeta .NET-ympäristössä, vaan npm-projektina (Node Package Manager). Käyttöliittymän lopullinen toteutus on delegoitu muulle Kooditar-tiimille. Tällä hetkellä toteutettuja käyttötapauksia on mm. kirjautuminen (kuva 27) ja projektien listaaminen käyttäjän oikeuksien perusteella.



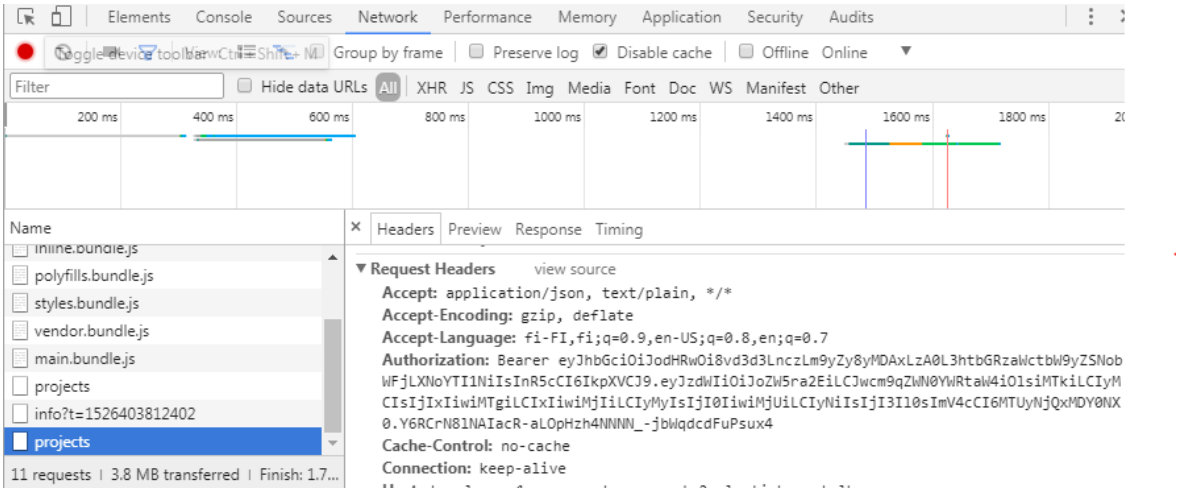
Kuva 27 Toteutettu kirjautumisnäkymä, kuvakaappaus

Kun käyttäjä kirjautuu, käyttöliittymäsovelluksen HTTP-pyyntöön tulee palvelimelta vastaus JSON-objektina, jossa on token-kentässä palvelimella generoitu JSON Web Token (ks. luku 5.1.4) (kuva 28).



Kuva 28 Palvelimelta palautunut HTTP-response autentikoinnin yhteydessä. Kuvakaappaus Chromen developer-työkalusta

Kun käyttöliittymäsovellus tekee muita kutsuja palvelimelle esimerkiksi hakeakseen projektit, joihin käyttäjällä on oikeudet, autentikoinnin yhteydessä saatu JSON Web Token asetetaan HTTP-kutsun Header-attribuutiksi.



Kuva 29 HTTP-kutsu ProjectsControlleriin, jossa yhtenä Header-attribuuttina on autentikoinnin yhteydsssä saatu JSON Web Token

Angular-sovellus pitää sisällään TypeScript-luokkia, jotka toteuttavat Angularin mukaisia module-, component- ja service-rakenteita. Edellä esitelty autenkointi on toteutettu AuthenticationService-nimisessä luokassa (kuva 30). AuthenticationService-luokka hyödyntää JavaScriptiin natiivisti kuuluvaa localStorage-ominaisuutta, jonka avulla selaimen muistiin voidaan tallentaa erilaisia arvoja. Kirjautumisen yhteydessä AuthenticationService tallentaa palvelimelta saadut käyttäjätiedot ja JSON Web Tokenin selaimen muistiin. Kun käyttäjä kirjautuu ulos, selaimen muistista poistetaan käyttäjän tiedot JWT mukaan lukien (ei näy kuvassa).

```
@Injectable()
export class AuthenticationService {
    public token: string;

    constructor(private http: Http) {

        var currentUser = JSON.parse(localStorage.getItem('currentUser'));
        this.token = currentUser && currentUser.token;
    }

    login(username: string, password: string): any {
        var data = {
            username: username,
            password: password
        }
        var headers = new Headers();
        headers.append('Content-Type', 'application/json');

        return this.http.post(globals.url + '/api/account/authenticate', data, { headers: headers })
            .map((response: Response) => {
                let token = response.json() && response.json().token;
                if (token) {
                    this.token = token;
                    localStorage.setItem('currentUser', JSON.stringify({ username: username, token: token }));
                    return true;
                }
                else {
                    return false;
                }
            });
    }
```

Kuva 30 Angularilla toteutettu AuthenticationService konstruktoreineen ja login-metodi

Projektit puolestaan haetaan palvelimelta ProjectService-luokasta käsin (kuva 31). Luokan konstruktorissa selaimen localStorage-muistista haetaan käyttäjätiedot ja JWT. Konstruktorissa asetetaan myös header-attribuutteihin localStoragesta haettu JWT. Luokan getProjects-metodi ottaa yhteyden palvelimelle ja liittää HTTP-kutsuun mukaan konstruktorissa asetetun header-attribuutin, jolloin palvelin pystyy vastaan ottamaan JWT:n ja tarkistamaan sen. Metodi palauttaa taulukon kaikista projekteista, joihin käyttäjällä on oikeudet.

```
@Injectable()
export class ProjectService {

    private token: string;
    private headers: Headers;


    constructor(private http: Http) {
        // set token if saved in local storage
        var currentUser = JSON.parse(localStorage.getItem('currentUser'));
        this.token = currentUser && currentUser.token;
        this.headers = new Headers();
        this.headers.append('Authorization', this.token);
    }


    public getProjects(): Observable<Project[]> {

        return this.http.get(globals.url + 'api/projects', { headers: this.headers })
            .map((response: Response) => {
                return response.json() as Project[];
            });
    }
```

Kuva 31 ProjectService-luokka, konstruktori ja getProjects-metodi

Datan esittäminen käyttäjälle tapahtuu komponenteissa, joihin servicet voidaan injektoida. HomeComponent-luokkaan (kuva 32) on injektoitu edellä esitelty ProjectService. Komponentti kutsuu käynnistyessään ngOnInit-metodia, joka hakee ProjectServicen getProjects-metodia hyödyntäen komponentin projects-kenttään käyttäjälle näkyvät projektit.

```
@Component({
    moduleId: module.id,
    templateUrl: 'home.component.html'
})

export class HomeComponent implements OnInit {

    projects: Project[];

    constructor(private projectService: ProjectService) {

    }

    ngOnInit() {
        this.projectService.getProjects()
            .subscribe(projects => {
                this.projects = projects;
            });
    }

}
```

Kuva 32 HomeComponent-luokka ja projektien haku ProjectServicen avulla

HomeComponent-luokkaan liittyy HTML-pohja home.component.html (kuva 33). HTML-pohjaan. HTML-pohja renderöi Angularin omaa syntaksia hyödyntäen komponentissa esiintyvän datan, eli projektit, HTML-dokumentin listaelementeiksi. Listausnäkymässä käyttäjälle näytetään projektin nimet.

```
<div class="row">
  <h2>Projektit</h2>
  <div class="row">
    <a [routerLink]="['/create-project']">Luo uusi</a>
  </div>
  <div class="row">
    <ul>
      <li *ngFor="let project of projects">
        <a [routerLink]="['/project/', project.id]">
        {{project.name}}</a>
    </li>
    </ul>

</div>
</div>
<div><p><a [routerLink]="['/login']">Logout</a></p></div>
```

Kuva 33 HomeComponentin html-pohja

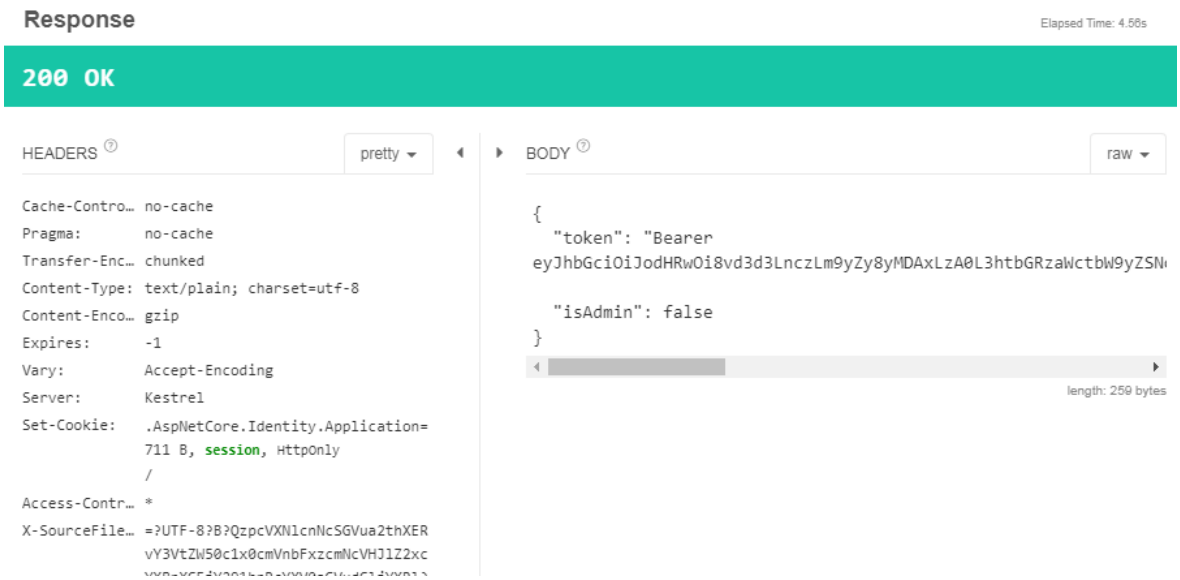
6.4 Testaus

Sovelluksen testauksen pohjana voidaan käyttää luvussa 6.1 esiteltyjä käyttäjätarinoita. Käyttäjätarinoita ei kuitenkaan vielä ole mahdollista testata käyttöliittymätasolla sen keskeneräisyyden vuoksi. Kaikkien käyttäjätarinoinen vaatimat api-reitit on kuitenkin toteutettu, ja niitä voidaan testata esimerkisi Postman-ohjelmalla tai Chrome-selaimeen asennettavalla Restlet-lisäosalla. Tässä luvussa esitellään muutama testitapaus Restletillä.

Kuvassa 34 on kuvakaappaus kirjautumisen testaamisesta Restletillä. HTTP-POST-pyyntö lähetetään localhost-osoitteeseen porttiin 51386. Tämä on osoite, jossa Visual Studio ajaa projektin paikallisesti. POST-pyynnön body-kenttään asetetaan JSON-objekti, jossa on käyttäjän käyttäjänimi ja salasana.

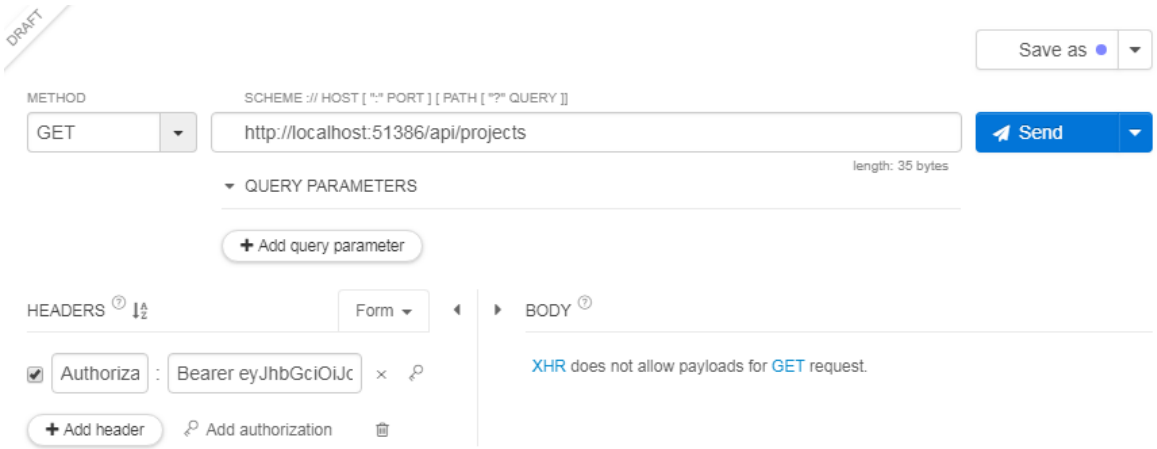
Kuva 34 Kirjautumisen testaus Restletillä

Mikäli pyyntö onnistuu, palvelimelta palautuu OK-vastaus (kuva 35), jossa on mukana muihin http-pyyntöihin vaadittava JSON Web Token.



Kuva 35 Palvelimen vastaus kirjautumispyyntöön

Kuvassa 36 haetaan palvelimelta projektit reitillä ”api/projects”. Pyyntö ohjautuu ProjectsControllerin Get-metodiin, joka on esitelty luvussa 6.3.1. Pyynnössä on header-attribuuttina kirjautumisen yhteydessä saatu JWT.

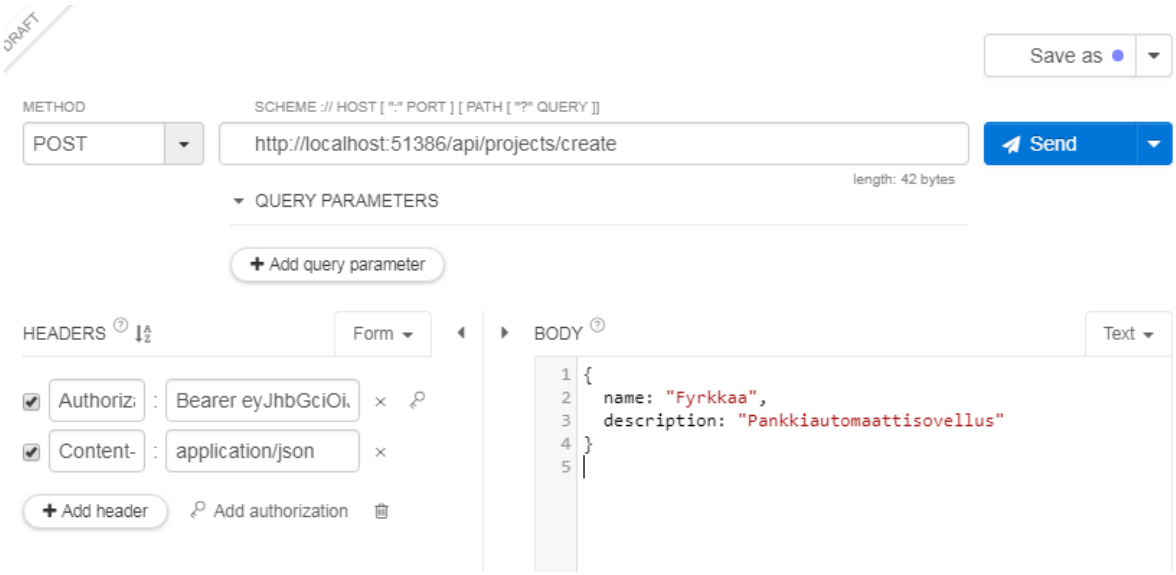


Kuva 36 Projektien haun testaus Restletillä

Pyynnön onnistuessa palvelimelta palautuu OK-vastaus sekä lista projekteista, joihin käyttäjällä on oikeudet (kuva 37). Listan olioilla on samat kentät kuin .NET-projektissa määritellyllä ProjectDTO-luokassa.

Kuva 37 Palvelimen vastaus projektien hakuun

Samalla tokenilla voidaan myös luoda järjestelmään uusi projekti (kuva 38). Projektille annetaan nimi ja kuvaus. Projektin luomisen onnistuessa palvelimelta palautuu tyhjä OK-vastaus.



Kuva 38 Uuden projektin luominen Restletillä

6.5 Julkaisu

Sovelluksen tietokanta ja REST-rajapinta on julkaistu Amazon-palvelimella. Käyttöliittymän julkaisusta päätetään myöhempänä ajankohtana. Tällä hetkellä käyttöliittymän kehitystä voidaan jatkaa joko siten, että koko projekti ajetaan paikallisesti, jolloin front end -projekti (käyttöliittymä) ottaa yhteyden localhost-osoitteen porttiin, jossa back end -projekti on ajettu, tai siten, että front end -projekti ottaa yhteyden Amazonin pilveen ja hyödyntää jo julkaistua back end -projektia.

7 Pohdinta

Projektin puitteissa on onnistuttu toteuttamaan Koodittaren projektinhallintasovelluksen runko, jonka ensimmäinen MVP-versio on käyttöliittymän luomista vaille valmis. Sovelluksen kehittäminen on mahdollistanut uusien teknologioiden omaksumisen yrityksessämme sekä synnyttänyt sille immateriaaliomaisuutta. Vielä ei ole näköpiirissä, minkälaisella aikautalulla sovellus mahdollisesti otetaan käyttöön.

Projektin etenemisessä on ollut haasteita aikataulun suhteen. Projektinhallintasovelluksen kehitykseltä on syönyt aikaa mm. Koodittaressa käynnissä olleet muut tuotekehityskokeilut, kuten erilaiset mobiilisovellukset.

Sovelluksen eriyttäminen selkeästi toisistaan riippumattomiin back-end- ja käyttöliittymärajapintoihin mahdollistaa paremman skaalautuvuuden ja lisää jatkokehitysmahdollisuuksia. Koska sovelluksessa on käytössä JSON Web Token -autentikointi sessioon perustuvan kirjautumisen sijaan, samaa kirjautumisjärjestelmää pystyttäisiin hyödyntämään esimerkiksi mobiilisovelluksessa.

Tämän opinnäytetyön aikana olen joutunut perehtymään syvällisemmin ohjelmistokehityksen eri osa-alueisiin pelkän ohjelmoinnin sijaan, mikä on parantanut kykyäni hahmottaa kokonaisuuksia. Valitut teknologiat ovat olleet minulle enimmäkseen entuudestaan tuttuja työelämästä, mutta uskon vahvistaneeni osaamistani niiden osalta kehitystyön aikana ilmi tulleiden uusien piirteiden myötä. Teknologioiden sopivuus yritykselleme jää joltain osain kyseenalaiseksi. Angularia ja erityisesti sen mobiiliviitekehystä Ionic Frameworkia olemme jo hyödyntäneet muissakin projekteissa. Sen sijaan tällä hetkellä käytössä olevat palvelimemme AWS:ää lukuunottamatta eivät tue .NET-teknologioita, joten niiden käyttö jää luultavasti vähäisemmäksi.

Lähteet

Angular - Architecture overview. Luettavissa: https://angular.io/guide/architecture. Luettu: 22.4.2018

Atlassian, 2017. What is a board? Luettavissa: confluence.atlassian.com/jirasoft-warecloud/what-is-a-board-764477964.html#Whatisaboard?-OnaScrumboard. Luettu: 15.11.2017

Barskiy, Sergey, 2015. Code-First Development with Entity Framework. ISBN 978-1-78439-627-5

Chauhan, Shailendra, 2013, päivitetty 2016. Luettavissa: www.dotnettricks.com/learn/webapi/difference-between-aspnet-mvc-and-aspnet-web-api. Luettu 8.12.2017

Chowdhuri, Shahed, 2016. ASP.NET Core Essentials. ISBN 978-1-78588-954-7

Cole, Rob & Scotcher, Edward 2015. Brilliant Agile Project Management. ISBN 978-1-292-06358-4

Flanagan, David, 2011. JavaScript : the definitive guide. ISBN 978-0-596-80552-4

Kanjilal, Joydip, 2013. ASP.NET Web API: Build RESTful web applications and services on the .NET framework

Li, Patrick, 2016. JIRA 7 Essentials – Fourth Edition. ISBN 1-78646-251-6

Michaelis, Mark, 2013. Essential C# 5.0. ISBN 0-321-87758-6

Microsoft, 16.6.2014. Create Data Transfer Objects (DTOs). Luettavissa: https://docs.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5. Luettu 13.5.2018

Nance, Christopher, 2014. TypeScript Essentials. ISBN 978-1-78398-577-7

Powell-Morse, Andrew, 2016. Iterative Model: What Is It And When Should You Use It? Luettavissa: https://airbrake.io/blog/sdlc/iterative-model. Luettu: 23.4.2018

Pot, Justin, 2011. Toggl Helps You Discover Where Your Time Goes. Luettavissa: https://www.makeuseof.com/tag/toggl-helps-discover-time. Luettu: 23.5.2018

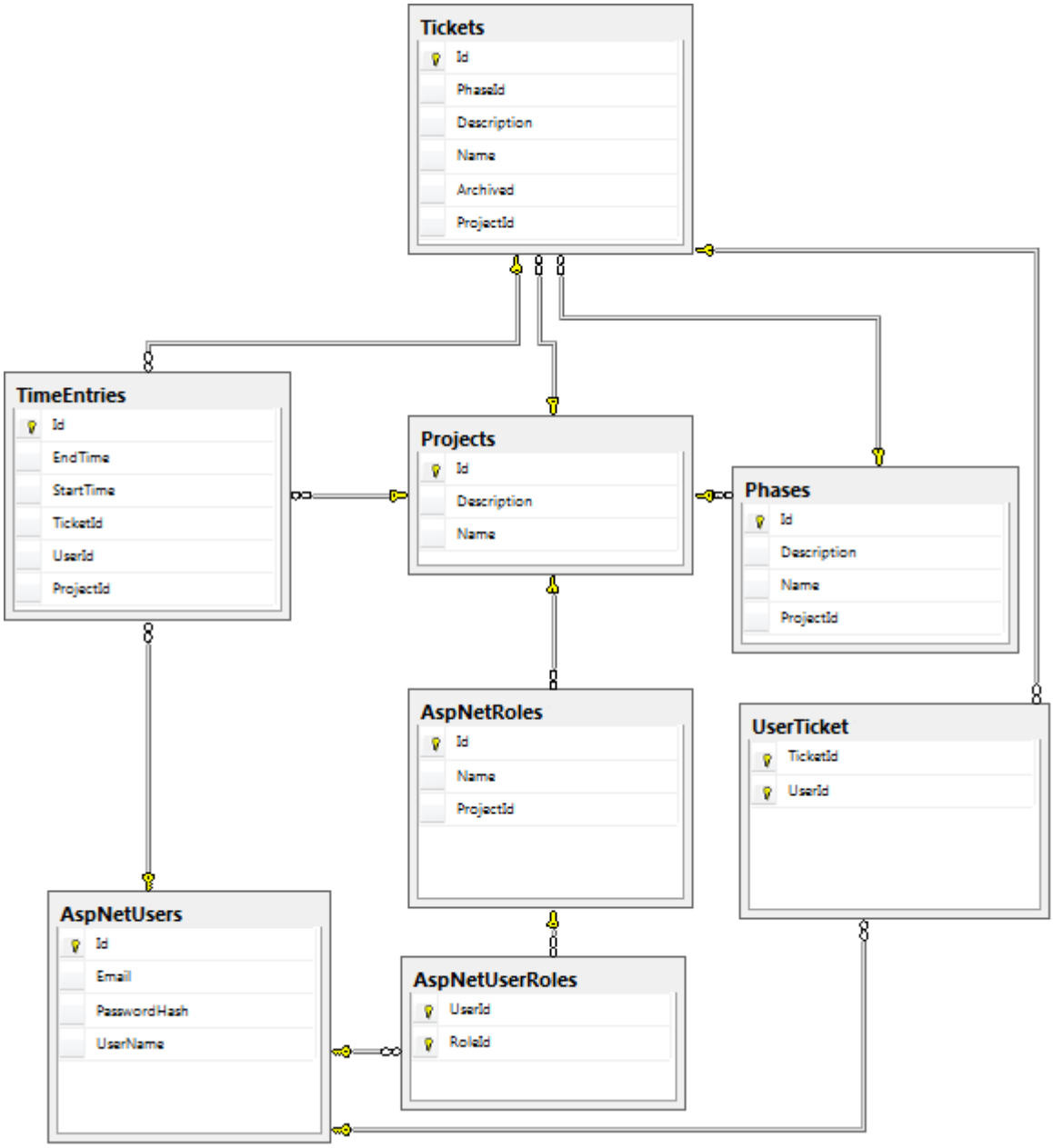
Troelsen, Andrew W. & Japikse, Philip, 2015. C# 6.0 and the .NET 4.6 Framework. ISBN 1-4842-1333-5

The Clever PM, 2016. Which is better – Kanban or Scrum? Luettavissa: www.cleverpm.com/2016/03/04/which-is-better-kanban-or-scrum/. Luettu: 8.12.2017

wpcurve.com, 2015. How we effectively use Trello for project management. Luettavissa: wpcurve.com/trello-for-project-management/. Luettu 14.11.2017

Liitteet

Liite 1. Tietokantakaavio



Liite 2. Tietokannan tietohakemistokuvaukset

Taulun nimi	**Projects**			
Määritelmä	Käyttäjän luoma projekti sovelluksessa			
Ominaisuudet	Nimi	**Kuvaus**	**Tietotyyppi**	**PK/FK/NOT NULL**
	Id	projektin yksilöllinen tunnus	int	PK, NOT NULL, AUTO-INCREMENT
	Name	projektin nimi	nvarchar(MAX)	NOT NULL
	Description	projektin kuvaus	nvarchar (MAX)	
PK	Id			

Esimerkkirivi

Id	1
Name	Tregl
Description	Projektinhallintasovellus

Taulun nimi	**Phases**			
Määritelmä	Projektiin liittyvä työvaihe.			
Ominaisuudet	Nimi	**Kuvaus**	**Tietotyyppi**	**PK/FK/NOT NULL**
	Id	työvaiheen yksilöllinen tunnus	int	PK, NOT NULL, AUTO-INCREMENT
	Name	työvaiheen nimi	nvarchar(MAX)	NOT NULL
	Description	työvaiheen kuvaus	nvarchar(MAX)	
	ProjectId	projektin tunnus, johon työvaihe liittyy	int	FK, NOT NULL
PK	Id			
FK	ProjectId			

Esimerkkirivi

Id	1
Name	To do
Description	NULL
ProjectId	1

Taulun nimi	**Tickets**

Määritelmä	Projektiin liittyvä työtehtävä.			
Ominaisuudet	Nimi	**Kuvaus**	**Tietotyyppi**	**PK/FK/NOT NULL**
	Id	työtehtävän yksilöllinen tunnus	int	PK, NOT NULL, AUTO-INCREMENT
	Name	työtehtävän nimi	nvarchar(MAX)	NOT NULL
	Description	työtehtävän kuvaus	nvarchar(MAX)	
	Archived	tieto siitä, onko työtehtävä arkistoitu	bit	NOT NULL
	ProjectId	projektin tunnus, johon työtehtävä liittyy	int	FK, NOT NULL
	PhaseId	sen työvaiheen tunnus, jossa työtehtävä tällä hetkellä on	int	FK, NOT NULL
PK	Id			
FK	ProjectId, PhaseId			

Esimerkkirivi

Id	1
Name	Tietokannan suunnittelu
Description	Projektille pitäisi saada tietokantamalli suunniteltua
Archived	0
ProjectId	1
PhaseId	1

Taulun nimi	**AspNetUsers**			
Määritelmä	ASP.NET Coren Identity-kirjaston toteutus sovelluksen käyttäjästä			
Ominaisuudet	Nimi	**Kuvaus**	**Tietotyyppi**	**PK/FK/NOT NULL**

	Id	käyttäjän yksilöllinen tunnus	nvarchar(450)	PK, NOT NULL
	Email	käyttäjän sähköpostiosoite	nvarchar(256)	NOT NULL
	UserName	käyttäjän käyttäjänimi	nvarchar(256)	NOT NULL
	PasswordHash	käyttäjän salasana SHA-256-algoritmilla salatussa muodossa	nvarchar(MAX)	NOT NULL
PK	Id			

Esimerkkirivi

Id	c724d704-ed25-4379-8850-2760a973b698
Email	henry.kari@kooditar.fi
UserName	Henry
PasswordHash	AQAAAAEAACcQAAAAEIGSEWR2hGAJfwQ+YahDoS3R0cS3aK7Hky8qcZbEntjZUnLhtV1QX/zUux3b50pkuA==

Taulun nimi	**TimeEntries**			
Määritelmä	Käyttäjän syöttämä aikamerkintä, joka kertoo ajankohdan, jolloin käyttäjä on aloittanut työskentelyn tietyn tehtävän parissa, ja ajankohdan, jolloin hän on päättänyt sen			
Ominaisuudet	Nimi	**Kuvaus**	**Tietotyyppi**	**PK/FK/NOT NULL**

	Id	aikamerkinnän yksilöllinen tunnus	int	PK, NOT NULL, AUTO-INCREMENT
	StartTime	aikamerkinnän alkamisajankohta	datetime(7)	NOT NULL
	EndTime	aikamerkinnän päättymisajankohta	datetime(7)	
	UserId	sen käyttäjän tunnus, joka on tehnyt aikamerkinnän	nvarchar(450)	FK, NOT NULL
	ProjectId	projektin tunnus, johon aikamerkintä liittyy	int	FK, NOT NULL
	TicketId	työtehtävän tunnus, johon aikamerkintä on liitetty	int	FK, NOT NULL
PK	Id			
FK	UserId, ProjectId, TicketId			

Esimerkkirivi

Id	1
StartTime	2017-11-17 08:00:00.0000000
EndTime	2017-11-17 16:00:00.0000000
UserId	c724d704-ed25-4379-8850-2760a973b698
ProjectId	1
TicketId	1

Taulun nimi	**AspNetRoles**			
Määritelmä	ASP.NET Coren Identity-kirjaston toteutus sovellukseen liittyvistä rooleista, joilla voi olla erilaisia käyttöoikeuksia sovelluksessa			
Ominaisuudet	Nimi	**Kuvaus**	**Tietotyyppi**	**PK/FK/NOT NULL**
	Id	roolin yksilöllinen tunnus	nvarchar(450)	PK, NOT NULL, AUTO-INCREMENT
	Name	roolin nimi	nvarchar(256)	NOT NULL
	ProjectId	rooliin liittyvä projekti	int	FK
PK	Id			
FK	ProjectId			

Esimerkkirivi

Id	04f6e816-56fb-42f6-bac9-d0954c4d3a9f
Name	projectadmin
ProjectId	1

Taulun nimi	**UserTicket**			
Määritelmä	Välitaulu, joka kuvastaa käyttäjän ja työtehtävän välistä suhdetta			
Ominaisuudet	Nimi	**Kuvaus**	**Tietotyyppi**	**PK/FK/NOT NULL**
	UserId	käyttäjän tunnus	nvarchar(450)	NOT NULL, PK, FK
	TicketId	työtehtävän tunnus	int	NOT NULL, PK, FK
PK	UserId, TicketId			
FK	UserId, TicketId			
Esimerkkirivi	Kts. alla			

Esimerkkirivi

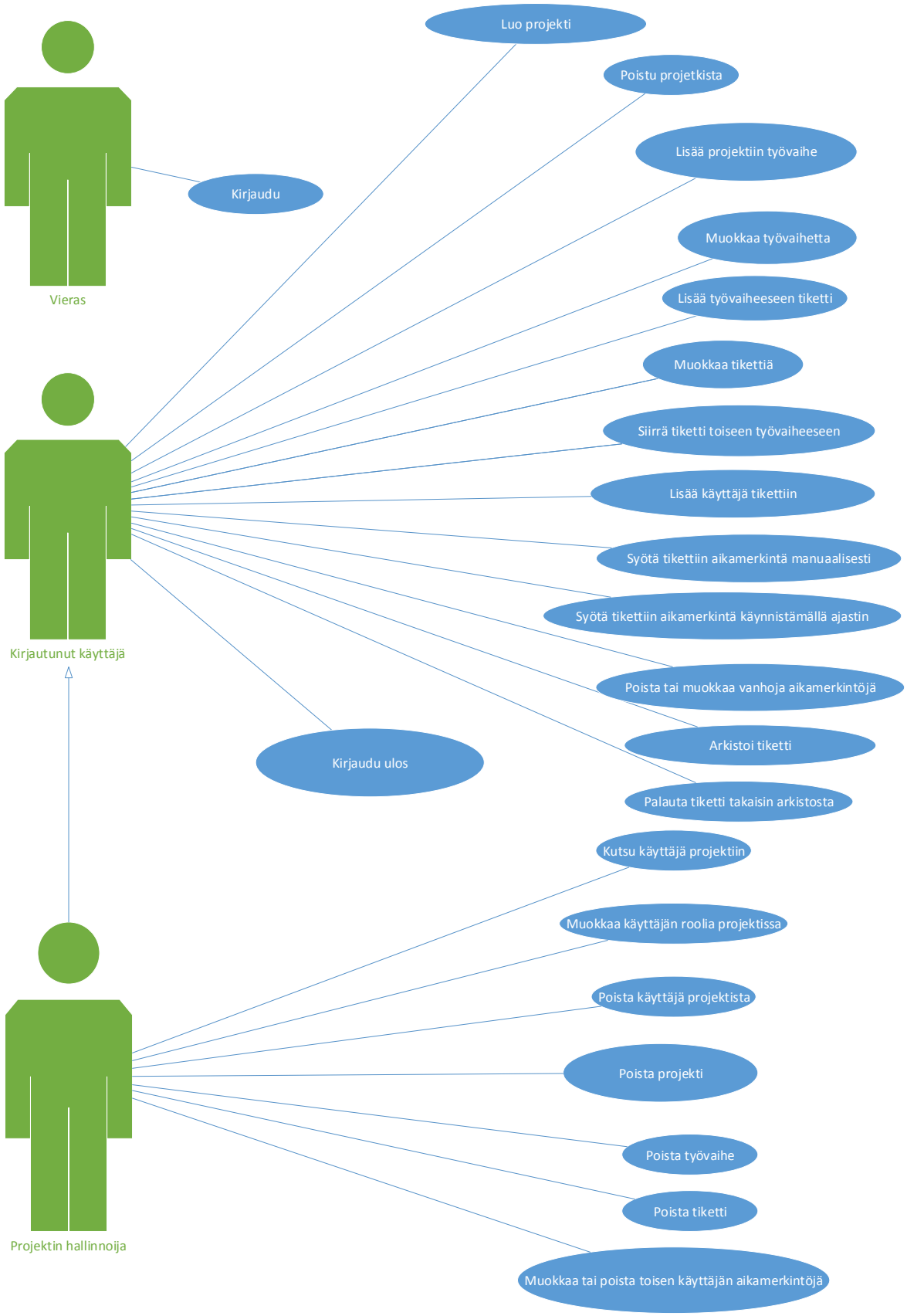
UserId	c724d704-ed25-4379-8850-2760a973b698
TicketId	1

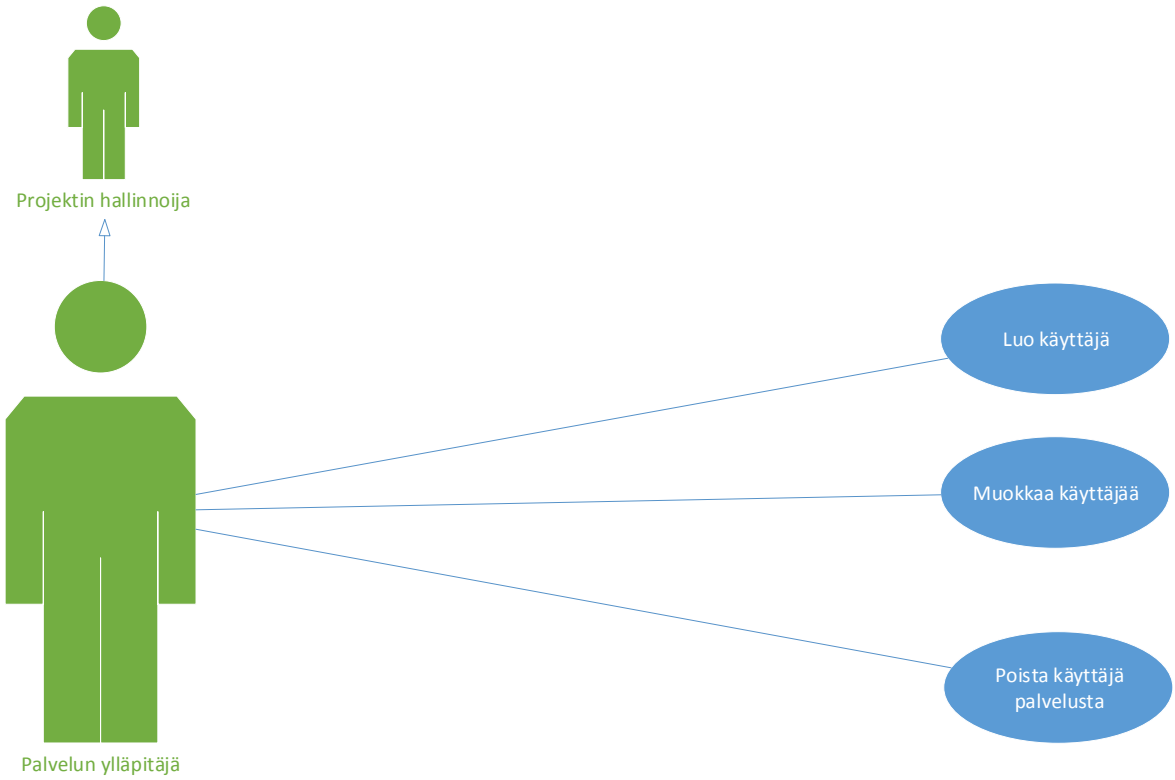
Taulun nimi	**AspNetUserRoles**			
Määritelmä	Välitaulu, joka kuvastaa käyttäjän ja käyttäjäroolin välistä suhdetta			
Ominaisuudet	Nimi	**Kuvaus**	**Tietotyyppi**	**PK/FK/NOT NULL**
	UserId	käyttäjän tunnus	nvarchar(450)	NOT NULL, PK, FK
	RoleId	roolin tunnus	nvarchar(450)	NOT NULL, PK, FK
PK	UserId, RoleId			
FK	UserId, RoleId			

Esimerkkirivi

UserId	c724d704-ed25-4379-8850-2760a973b698
RoleId	04f6e816-56fb-42f6-bac9-d0954c4d3a9f

Liite 3. Käyttötapauskaavio




Projektin hallinnoija
Palvelun ylläpitäjä
Luo käyttäjä
Muokkaa käyttäjää
Poista käyttäjä palvelusta

Liite 4. REST-esimerkkejä

Projektien listaus

Endpoint	**/api/projects**
Metodi	HTTP GET
Request-esimerkki	{"Authorization": "Bearer eyJhbGciOiJodH-RwOi8vd3d3LnczL … -II_HDek"}

Success-esimerkki	HTTP/1.1 200 Ok
	[
	{
	"id": 19,
	"name": "Fyrkkaa",
	"description": "Pankkiautomaattisovellus",
	"users": null,
	"totalTime": "00:00:00"
	},
	{
	"id": 20,
	"name": "Hakia",
	"description": "Raksavuokratyöfirma",
	"users": null,
	"totalTime": "00:00:00"
	}
	]

Error-esimerkki	HTTP/1.1 401 Unauthorized

Projektin luominen

Endpoint	**/api/projects/create**
Metodi	HTTP POST
Request-esimerkki	{"Authorization": "Bearer eyJhbGciOiJodH-RwOi8vd3d3LnczL … -II_HDek"}
Json payload	{"name":"Safkaa", "description":"Pikaruokamobiili-sovellus"}
Success-esimerkki	HTTP/1.1 200 Ok
Error-esimerkki	HTTP/1.1 401 Unauthorized

Projektin työvaiheiden listaus

Endpoint	**/api/projects/:projectId/phases**
Metodi	HTTP GET
Parametrit	projectId: number
Request-esimerkki	{"Authorization": "Bearer eyJhbGciOiJodH-RwOi8vd3d3LnczL … -II_HDek"}

Success-esimerkki	HTTP/1.1 200 Ok
	[
	{
	"id": 19,
	"name": "To do",
	"description": "Aloitettavat työt",
	},
	{
	"id": 20,
	"name": "In progress",
	"description": "Keskeneräiset työt",
	},
	{
	"id": 21,
	"name": "Done",
	"description": "Valmiit työt",
	}
	]

Error-esimerkki	HTTP/1.1 401 Unauthorized

Projektin tikettien listaus

Endpoint	**/api/projects/:projectId/tickets**
Metodi	HTTP GET
Parametrit	projectId: number
Request-esimerkki	{"Authorization": "Bearer eyJhbGciOiJodH-RwOi8vd3d3LnczL … -II_HDek"}
Success-esimerkki	HTTP/1.1 200 Ok [{ "id": 1, "name": "Projektinäkymä",

	"description": "Projektin dashboard, jossa näkyy työvaiheet ja tiketit",
	"archived": false,
	"projectId": 1,
	"phaseId": 1,
	"users":
	["santeri", "panu"],
	"totalTime": 00:00:00,
	},
	{
	"id": 2,
	"name": "Aikamerkinnän lisääminen",
	"description": "Käyttäjä pystyy merkitsemään johonkin tikettiin käyttämänsä ajan",
	"archived": false,
	"projectId": 1,
	"phaseId": 1,
	"users":
	["santeri", "kolu"],
	"totalTime": 00:00:00,
	},
	{
	"id": 3,
	"name": "API:n dokumentointi",
	"description": "Apidocin luominen REST-rajapinnasta",
	"archived": false,
	"projectId": 1,
	"phaseId": 2,
	"users":
	["henkka"],
	"totalTime": 00:00:00,
	}
	]

Error-esimerkki	HTTP/1.1 401 Unauthorized

Tiketin luominen

Endpoint	**/api/projects/:projectId/tickets/create**
Metodi	HTTP POST
Parametrit	projectId: number
Request-esimerkki	{"Authorization": "Bearer eyJhbGciOiJodH-RwOi8vd3d3LnczL … -II_HDek"}
Json payload	{ "name":"Aikamerkinnän muokkaus", "description": "Toteuta ominaisuus, jolla käyttäjä pystyy muokkaamaan vanhaa aikamerkintäänsä",

	"phaseId": 1 }
Success-esimerkki	HTTP/1.1 200 Ok
Error-esimerkki	HTTP/1.1 401 Unauthorized

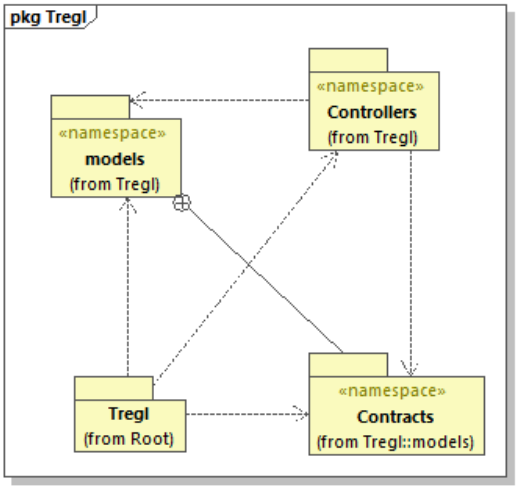
Aikamerkinnän luominen

Endpoint	**/api/projects/:projectId/timeentries/create**
Metodi	HTTP POST
Parametrit	projectId: number
Request-esimerkki	{"Authorization": "Bearer eyJhbGciOiJodH- RwOi8vd3d3LnczL … -II_HDek"}

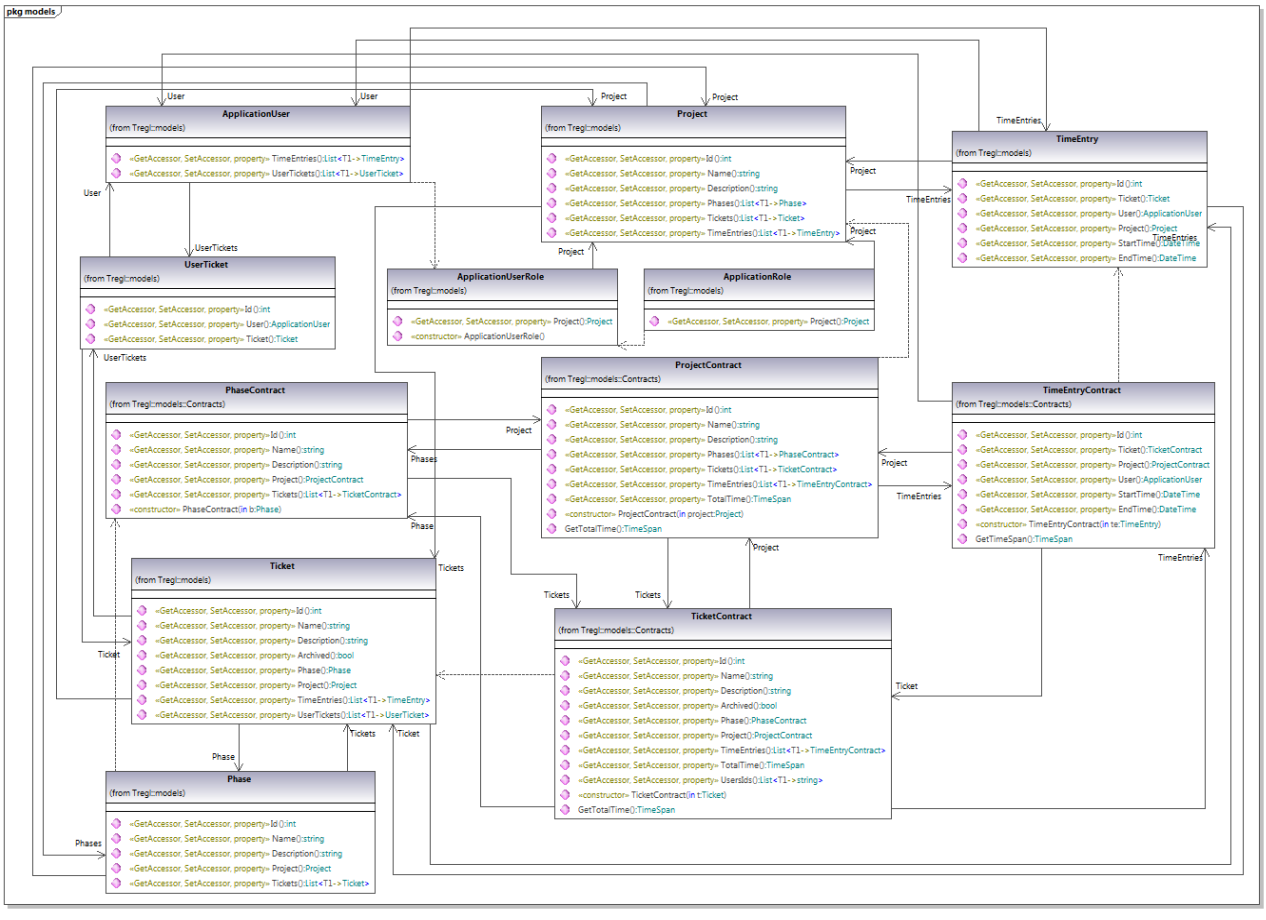
Json payload	{
	"ticketId": 1
	"startTime": "2017-12-01 08:00:00",
	"endTime": "2017-12-01 18:39:43"
	}

Success-esimerkki	HTTP/1.1 200 Ok
Error-esimerkki	HTTP/1.1 401 Unauthorized

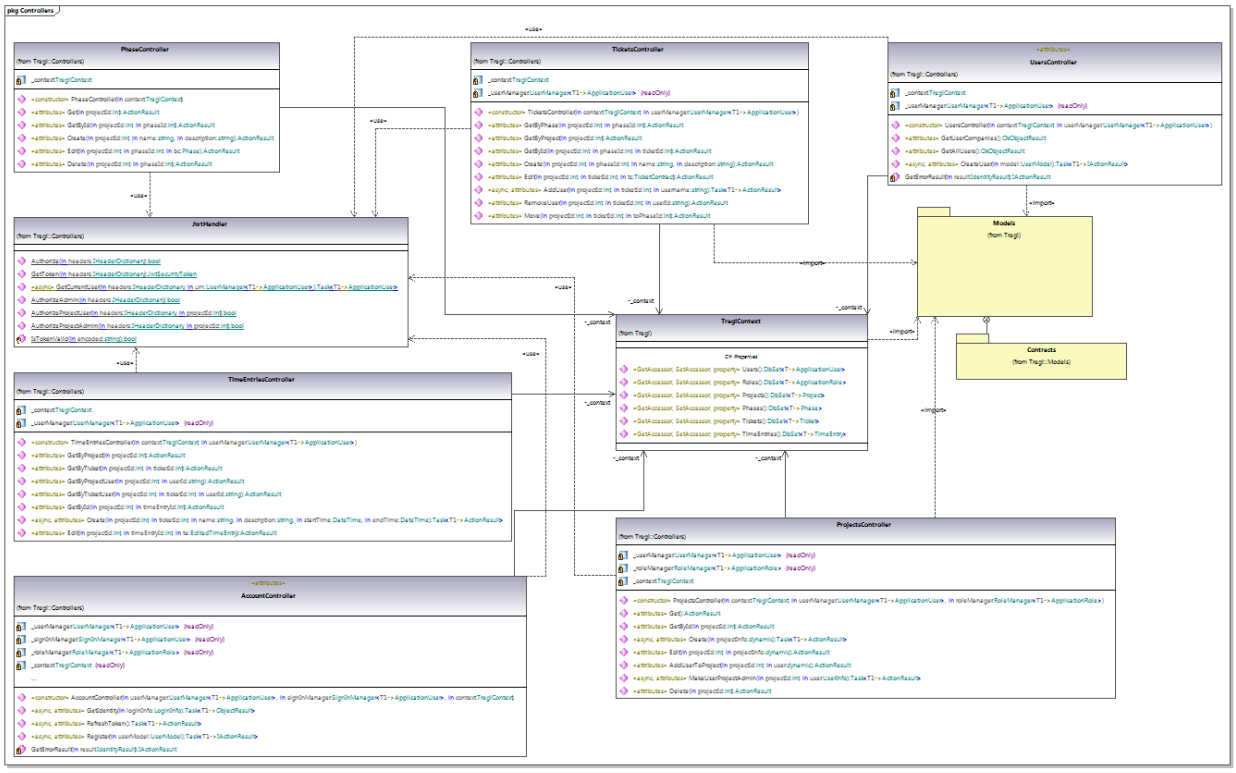
Liite 5. Toteutuksen luokkakaaviot



Generated by UModel www.altova.com



Generated by UModel www.altova.com


pkg Controllers
PhaseController
(from TrelloClone::Controllers)
_contextTrelloCloneContext
+constructor+ PhaseController(in contextTrelloCloneContext)
+attributes+ GetById(in projectId:int) in phaseId:int):ActionResult
+attributes+ Create(in projectId:int) in name:string, in description:string):ActionResult
+attributes+ Edit(in projectId:int) in phaseId:int) in bc:Phase):ActionResult
+attributes+ Delete(in projectId:int) in phaseId:int):ActionResult
TicketsController
(from TrelloClone::Controllers)
_contextTrelloCloneContext
_userManagerUserManagerT1->ApplicationUser> {readOnly}
+constructor+ TicketsController(in contextTrelloCloneContext) in userManagerUserManagerT1->ApplicationUser>)
+attributes+ GetByPhase(in projectId:int) in phaseId:int):ActionResult
+attributes+ GetByProject(in projectId:int):ActionResult
+attributes+ GetById(in projectId:int) in phaseId:int) in ticketId:int):ActionResult
+attributes+ Create(in projectId:int) in phaseId:int) in name:string, in description:string):ActionResult
+attributes+ Edit(in projectId:int) in ticketId:int) in tc:TicketContext):ActionResult
+async, attributes+ AddUser(in projectId:int) in ticketId:int) in username:string):TaskT1->ActionResult>
+attributes+ RemoveUser(in projectId:int) in ticketId:int) in userId:string):ActionResult
+attributes+ Move(in projectId:int) in ticketId:int) in toPhaseId:int):ActionResult
«attributes»
UsersController
(from TrelloClone::Controllers)
_contextTrelloCloneContext
_userManagerUserManagerT1->ApplicationUser> {readOnly}
+constructor+ UsersController(in contextTrelloCloneContext) in userManagerUserManagerT1->ApplicationUser>)
+attributes+ GetUserCompanies():OkObjectResult
+attributes+ GetAllUsers():OkObjectResult
+async, attributes+ CreateUser(in model:UserModel):TaskT1->IActionResult>
GetErrorResult(in result:IdentityResult):IActionResult
JwtHandler
(from TrelloClone::Controllers)
AuthorizeUser(in headers:IHeaderDictionary):bool
GetToken(in headers:IHeaderDictionary):JwtSecurityToken
+async, GetCurrentUser(in headers:IHeaderDictionary, in um:UserManagerT1->ApplicationUser>):TaskT1->ApplicationUser>
AuthorizeAdmin(in headers:IHeaderDictionary):bool
AuthorizeProjectUser(in headers:IHeaderDictionary, in projectId:int):bool
AuthorizeProjectAdmin(in headers:IHeaderDictionary, in projectId:int):bool
isTokenValid(in encoded:string):bool
Models
(from TrelloClone::Models)
Contracts
(from TrelloClone::Models)
TrelloCloneContext
(from TrelloClone)
C# Properties
+GetAccessor, SetAccessor, property+ Users():DbSetT->ApplicationUser>
+GetAccessor, SetAccessor, property+ Roles():DbSetT->ApplicationRole>
+GetAccessor, SetAccessor, property+ Projects():DbSetT->Project>
+GetAccessor, SetAccessor, property+ Phases():DbSetT->Phase>
+GetAccessor, SetAccessor, property+ Tickets():DbSetT->Ticket>
+GetAccessor, SetAccessor, property+ TimeEntries():DbSetT->TimeEntry>
TimeEntriesController
(from TrelloClone::Controllers)
_contextTrelloCloneContext
_userManagerUserManagerT1->ApplicationUser> {readOnly}
+constructor+ TimeEntriesController(in contextTrelloCloneContext) in userManagerUserManagerT1->ApplicationUser>)
+attributes+ GetByProject(in projectId:int):ActionResult
+attributes+ GetByTicket(in projectId:int) in ticketId:int):ActionResult
+attributes+ GetByProjectUser(in projectId:int) in userId:string):ActionResult
+attributes+ GetByTicketUser(in projectId:int) in ticketId:int) in userId:string):ActionResult
+attributes+ GetById(in projectId:int) in timeEntryId:int):ActionResult
+async, attributes+ Create(in projectId:int) in ticketId:int) in name:string, in description:string, in startTime:DateTime, in endTime:DateTime):TaskT1->ActionResult>
+attributes+ Edit(in projectId:int) in timeEntryId:int) in te:EditedTimeEntry):ActionResult
ProjectsController
(from TrelloClone::Controllers)
_userManagerUserManagerT1->ApplicationUser> {readOnly}
_roleManagerRoleManagerT1->ApplicationRole> {readOnly}
_contextTrelloCloneContext
+constructor+ ProjectsController(in contextTrelloCloneContext) in userManagerUserManagerT1->ApplicationUser>, in roleManagerRoleManagerT1->ApplicationRole>)
+attributes+ Get():ActionResult
+attributes+ GetById(in projectId:int):ActionResult
+async, attributes+ Create(in projectInfo:dynamic):TaskT1->ActionResult>
+attributes+ Edit(in projectId:int) in projectInfo:dynamic):ActionResult
+attributes+ AddUserToProject(in projectId:int) in userId:dynamic):ActionResult
+async, attributes+ MakeUserProjectAdmin(in projectId:int) in userUserInfo):TaskT1->ActionResult>
+attributes+ Delete(in projectId:int):ActionResult
«attributes»
AccountController
(from TrelloClone::Controllers)
_userManagerUserManagerT1->ApplicationUser> {readOnly}
_signInManagerSignInManagerT1->ApplicationUser> {readOnly}
_roleManagerRoleManagerT1->ApplicationRole> {readOnly}
_contextTrelloCloneContext {readOnly}
...
+constructor+ AccountController(in userManagerUserManagerT1->ApplicationUser>, in signInManagerSignInManagerT1->ApplicationUser>, in contextTrelloCloneContext)
+async, attributes+ GetIdentity(in loginInfo:LoginInfo):TaskT1->ObjectResult>
+async, attributes+ RefreshToken():TaskT1->ActionResult>
+async, attributes+ Register(in userModel:UserModel):TaskT1->IActionResult>
GetErrorResult(in result:IdentityResult):IActionResult
+use+
+import+
_context
Generated by UModel
www.altova.com