WEB-SOVELLUKSEN TEKO DJANGOLLA

Ammattikorkeakoulututkinnon opinnäytetyö

Riihimäki, Tietotekniikan koulutusohjelma

Kevät, 2018

Tommi Vuori

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietotekniikan koulutusohjelma
Riihimäki

Tekijä Tommi Vuori **Vuosi** 2018

Työn nimi Web-sovelluksen teko Djangolla

Työn ohjaaja/t Petri Kuittinen

TIIVISTELMÄ

Tässä opinnäytetyössä tutustuttiin web-sovelluksien kehittämiseen tarkoitettuun Django-sovelluskehykseen ja sen käyttämään Python-ohjelmointikieleen. Näiden lisäksi työssä käytiin läpi yleisesti mitä web-sovellukset ovat ja mitä ominaisuuksia web-sovelluskehykset tavallisesti sisältävät.

Projektina toteutettiin esimerkkisovellus Djangolla. Sovelluksena toimi ominaisuuksiltaan melko rajattu linkin lyhentämiseen tarkoitettu web-sovellus. Sovelluksen visuaalisen ilmeen toteutukseen käytettiin Bootstrap-kirjastoa. Ennen opinnäytetyötä olin käynyt läpi muutamia tutoriaaleja Djangolla ja tutustunut myös muihin sovelluskehyksiin. Minulla oli myös hieman kokemusta Bootstrap-kirjastosta.

Opinnäytetyön tuloksena saatiin tehtyä responsiivinen linkin lyhentämiseen tarkoitettu web-sovellus. Valmis sovellus oli ominaisuuksiltaan melko rajattu sisältäen vain sovelluksen toiminnan kannalta oleellisimmat ominaisuudet. Linkin lyhentämisen lisäksi sovelluksesta löytyi tuki käyttäjätileille ja linkkien hallintaan tarkoitettu hallintapaneeli. Jatkokehitystä ajatellen sovellus oli helposti laajennettavissa isommaksi kokonaisuudeksi.

Avainsanat Django, Python, web-sovellus, web-sovelluskehys

Sivut 44 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information Technology
Riihimäki

Author Tommi Vuori **Year** 2018

Subject Creating a web application with Django

Supervisors Petri Kuittinen

ABSTRACT

The focus of this thesis was on the Django web framework and the Python programming language it uses. In addition to these, the thesis takes a general look at what web applications are and what features are usually included in the current popular web frameworks.

As the practical part of the thesis a link shortening application was created using Django. Bootstrap was used for the visual interface of the application. Prior to the thesis I had gone through a couple of Django tutorials and introduced myself to other popular frameworks. I also had some experience with Bootstrap.

The result of the thesis was a working and responsive link shortening application. The application was quite limited in functionality including only the most important features. In addition to link shortening, the application has support for user accounts and contains a link management dashboard. The current application would be easy to extend into a more complete and bigger application with further development.

Keywords Django, Python, web application, web framework

Pages 44 pages

SANASTO

CSS	Cascading Style Sheets	Web-sivujen ulkoasun ja visuaalisen ilmeen määrittämiseen tarkoitettu kieli.
DOM	Document Object Model	Malli, jonka avulla HTML-sivua on mahdollista muokata JavaScriptilla.
HTML	Hypertext Markup Language	Merkintäkieli, jolla määritetään sivuston rakenne ja sisältö.
HTTP	Hypertext Transfer Protocol	Protokolla, jota selaimet ja palvelimet käyttävät tiedonsiirtoon.
JSX	JavaScript XML	Laajennus tavallisen JavaScript-ohjelmointikielen syntaksiin.
MVC	Model-View-Controller	Ohjelmistoarkkitehtuuri, jonka tarkoituksena on erottaa malli, näkymä ja käsittelijä toisistaan.
MVT	Model-View-Template	Djangon versio suositusta MVC-arkkitehtuurista.
ORM	Object-relational mapping	Tekniikka, jonka avulla tietokantoja voi käyttää oliomaisesti.
REST	Representational State Transfer	HTTP-protokollaan perustuva tapa rajapintojen toteuttamiseen.
SPA	Single-Page Application	Sovellus, joka toimii usein vain yhdellä sivun latauksella. Data ja sisältö ladataan ja esitetään dynaamisesti.
SQL	Structured Query Language	Kyselykieli, jota käytännössä kaikki relaatiotietokannat käyttävät.
XML	Extensible Markup Language	Tietynlaisten merkintäkielten standardi.

SISÄLLYS

1 JOHDANTO

Tässä opinnäytetyössä tutustutaan web-sovelluksen rakentamiseen Python-ohjelmointikieltä käyttävällä Django-sovelluskehyksellä. Työn alussa käydään läpi yleisesti mitä web-sovelluksella ja sovelluskehyksellä tarkoitetaan, ja mitä ominaisuuksia eri sovelluskehykset tavallisesti sisältävät. Tämän jälkeen tutustutaan tämän hetken suosituimpiin web-sovelluskehyksiin. Sitten työssä perehdytään Python-ohjelmointikieleen ja tarkemmin Djangon ominaisuuksiin ja toimintaperiaatteisiin. Projektiosuuden alussa käydään läpi projektissa käytettävät työkalut ja kehitysympäristö.

Opinnäytetyön tavoitteena oli luoda käyttövalmis pitkien linkkien lyhentämiseen keskittynyt web-sovellus Djangolla. Sovelluksen vaatimuksina oli linkkien lyhentämisen lisäksi tuki käyttäjätilien tekoon ja niiden hallinnointiin. Opinnäytetyössä dokumentoitiin sovelluksen keskeisimmät ominaisuudet. Työn lopussa pohditaan mahdollisia jatkokehitykseen liittyviä ideoita ja käydään läpi työssä ilmenneitä ongelmia.

Opinnäytetyö on suunnattu henkilöille, jotka ovat kiinnostuneita web-sovellusten kehityksestä tai yleisesti web-sovelluskehyksistä. Opinnäytetyö olettaa lukijalta jonkin tasoista ymmärrystä ohjelmoinnin perusteista.

2 WEB-SOVELLUKSET

Web-sovelluksella tarkoitetaan sovellusta, joka suoritetaan käyttäen webselainta. Tavallisesta www-sivusta web-sovellus eroaa siten, että se sisältää dynaamista ja käyttäjän syötteen perusteella muuttuvaa dataa. Websovellus koostuu tavallisesti käyttäjän selaimessa suoritettavasta HTML, CSS – ja JavaScript-koodista ja se keskustelee palvelimella olevan koodin kanssa. Web-sovellus voi toimia myös kokonaan ilman palvelinta. Esimerkki tällaisesta sovelluksesta olisi JavaScriptilla tehty laskin, jonka ominaisuudet eivät tarvitse palvelimella olevaa tietokantaa tai laskentatehoa. (Ndegwa 2016.)

Web-sovelluksen HTML-rakenne ja sisältö voidaan renderöidä joko palvelimella tai vasta käyttäjän selaimessa. Renderöinnillä tarkoitetaan käyttäjän selaimessa lopulta näkyvän HTML-sivuston rakentamista yhdestä tai useammasta osasta. Kummallakin tavalla on omat hyödyt ja haitat, ja sovelluksen käyttötarkoitus määrittää yleensä sen, kumpaa tapaa kannattaa käyttää. (Ndegwa 2016.)

Yleisin tapa on rakentaa valmis HTML-dokumentti kokonaan palvelimella, jonka jälkeen se lähetetään sellaisenaan käyttäjälle. Tämä tapa soveltuu

hyvin sivustoille ja sovelluksille, jotka eivät sisällä isoja määriä dynaamista sisältöä. Tällaiset sivustot on myös helpompi hakukoneoptimoida, koska tavallisimmat hakukoneiden käyttämät botit eivät pysty renderöimään sisältöä paikallisesti. (Vega 2017.)

Kun sivun HTML on tarkoitus renderöidä pääasiassa selaimella, palvelin lähettää vain pienen pätkän HTML-koodia JavaScript-koodin lisäksi. Loput HTML:stä rakennetaan vasta käyttäjän selaimessa JavaScript-koodin avulla. JavaScript-koodin ja sen lähettämien palvelinkutsujen ajamisessa kestää hetken, jonka takia dynaamisen sisällön tilalla näkyy usein placeholder-sisältöä. (Vega 2017.)

3 WEB-SOVELLUSKEHYKSET

Sovelluskehyksellä tarkoitetaan ohjelmistoa ja kirjastoja, jotka muodostavat rungon sen päälle rakennettavalle sovellukselle. Sovelluskehys ei siis ole valmis käytettävissä oleva ohjelma, vaan toimii ainoastaan pohjana sisältäen yleisimmin tarvittavat ominaisuudet. Sovelluskehyksen tärkeimpänä tehtävänä on vähentää usein käytettyjen ominaisuuksien ohjelmointia uudelleen, eli ne sisältävät valmiiksi perustoiminnallisuudet ja näin lyhentävät sovelluksen kehitykseen menevää aikaa merkittävästi. (Niemi 2015.)

Web-sovelluskehykset ovat nimensä mukaisesti web-sovelluksien tekoon tarkoitettuja sovelluskehyksiä. Web-sovelluskehyksiä on tehty lähes jokaiselle suositulle ohjelmointikielelle matalan tason C++-kielestä korkean tason C#-kieleen. Tällä hetkellä yleisimmät web-sovelluksien tekoon käytettävät kielet ovat C#, PHP, Java, JavaScript, Python ja Ruby. Web-sovelluskehykset voidaan jakaa kahteen eri kategoriaan sen perusteella toimivatko ne asiakaspuolella vai palvelinpuolella. (Niemi 2015.)

3.1 Mikrosovelluskehykset

Massiivisten sovelluskehysten lisäksi on olemassa myös pienempiä sovelluskehyksiä, jotka sisältävät hyvin vähän sisäänrakennettuja toimintoja ja ominaisuuksia. Näitä käyttäen myös projektin rakenne on vapaampi, jonka ansiosta ohjelmoijalla on vapaammat kädet sovelluksen suhteen. Yksi esimerkki tällaisesta sovelluskehyksestä on Python-pohjainen Flask, jossa lähes jokainen ominaisuus täytyy tehdä itse tai asentaa erikseen laajennuksena. Myös erittäin suosittu Node.js-pohjainen Express.js lukeutuu samaan kategoriaan. (Ronacher 2017.)

3.2 Suosituimmat web-sovelluskehykset

Perehdyin eri web-sovelluskehysten suosioon. HotFrameworks on sivusto, joka mittaa eri sovelluskehysten suosiota niiden GitHub- ja Stack Overflow -aktiivisuuden perusteella. Microsoftin sovelluskehysten GitHub-pisteet kuitenkin puuttuivat kokonaan, jonka takia niiden todellinen suosio on todennäköisesti HotFrameworks-sivuston tuloksia pienempi (Kuva 1). Muilta osin pidin HotFrameworks-sivuston tuloksia luotettavina. (HotFrameworks 2018.)

Framework	Github Score	Stack Overflow Score	Overall Score
ASP.NET		100	100
AngularJS	95	97	96
Ruby on Rails	92	98	95
ASP.NET MVC		94	94
React	100	89	94
Django	90	94	92
Angular	91	91	91
Laravel	92	89	90
Spring	86	92	89
Express	91	83	87
Vue.js	99	76	87

Kuva 1. Suosituimmat web-sovelluskehykset HotFrameworks-sivuston mukaan.

3.3 Asiakaspuolen web-sovelluskehykset

Asiakaspuolen web-sovelluskehyksillä rakennetaan pääasiassa selaimessa toimivia JavaScript-pohjaisia sovelluksia. Ne tavallisesti keskittyvät dynaamisen datan käsittelyn helpottamiseen ja sen näyttämiseen HTML-sivulla. Palvelimien ja tietokantojen kanssa keskustelemiseen asiakaspuolen sovellukset käyttävät REST-rajapintoja. Käytännössä kaikki asiakaspuolen web-sovelluskehykset käyttävät JavaScriptia tai jotain siihen pohjautuvaa kieltä.

3.3.1 AngularJS ja Angular

AngularJS ja Angular ovat Googlen ylläpitämiä avointa lähdekoodia olevia asiakaspuolen sovelluksien rakentamiseen tarkoitettuja sovelluskehyksiä.

AngularJS-nimellä tarkoitetaan vanhempaa 1.x-versiota ja pelkällä Angular-nimellä tarkoitetaan uudelleenkirjoituksen jälkeisiä versioita. (Angular n.d.)

AngularJS on Miško Heveryn kehittämä sovelluskehys, joka oli aluksi vain apuna Googlen sisäisissä projekteissa. Sovelluskehyksen ensimmäinen julkinen versio julkaistiin vuonna 2010 avoimen lähdekoodin projektina. AngularJS oli kuitenkin ominaisuuksiltaan rajattu, eikä se soveltunut kunnolla isojen sovellusten rakentamiseen. (Gavigan 2018.)

Sovelluskehykselle tehtiin kokonainen uudelleenkirjoitus, josta vastasi Google. Uusi versio julkaistiin vuonna 2016 versionumerolla 2.0. Uudelleenkirjoituksesta johtuen syntaksi ja sovellusten rakenne oli erilainen. Se ei myöskään ollut taaksepäin yhteensopiva vanhempien AngularJS-sovellusten kanssa. Uudelleenkirjoitettu versio käytti oletuksena JavaScriptin sijasta TypeScriptia, joka lisää JavaScriptiin uusia ominaisuuksia. (Gavigan 2018.)

3.3.2 React

React on Facebookin ylläpitämä JavaScript-pohjainen käyttöliittymien tekoon tarkoitettu sovelluskehys. Ensimmäinen julkinen versio Reactista julkaistiin vuonna 2013. Web-sovellusten lisäksi Reactia voi käyttää mobiilisovellusten tekoon. Mobiilisovellusten tekoon tarkoitettu kirjasto on nimeltään React Native.

Virtuaalisen DOM:in ansiosta dynaamisen ja muuttuvan datan esittäminen on erittäin nopeaa. Kun ainoastaan yhtä sivun elementtiä muutetaan, riittää, että ainoastaan kyseinen elementti renderöidään uudelleen. Tavallista DOM:ia käytettäessä koko sivu pitäisi renderöidä uudelleen. (Reactjs n.d.)

React-sovellukset rakentuvat komponenteista. Komponentit käyttävät JSX-syntaksia, jonka avulla voi sekoittaa JavaScript- ja HTML-koodia keskenään. Tämän avulla voi esimerkiksi asettaa funktion palauttamaan HTML-elementtejä, mikä helpottaa ja nopeuttaa kehitystä. (Reactjs n.d.)

3.3.3 Vue

Vue on avointa lähdekoodia oleva asiakaspuolen sovelluksien kehittämiseen tarkoitettu sovelluskehys. Vuen mukana tuleva ydinkirjasto keskittyy pääasiassa näkymien rakentamiseen, jonka takia sitä on helppo käyttää myös muiden sovelluskehysten ja tekniikoiden kanssa. Pelkällä Vuella voi myös tehdä kokonaisia SPA-sovelluksia, mutta se vaatii ydinkirjaston lisäksi muita kirjastoja. Muut kirjastot hoitavat esimerkiksi URL-reitit ykset.
(Vuejs n.d. a.)

Ominaisuuksiltaan Vue on samankaltainen Reactin kanssa. Molemmat hyödyntävät virtuaalista DOM:ia ja sovelluksen eri osat rakentuvat komponenteista. Vue tukee myös tarvittaessa Reactista tuttua JSX-syntaksia, vaikkakin oletuksena käytössä on yksinkertaisempi HTML-pohjainen sivupohja-järjestelmä. (Vuejs n.d. b.)

3.4 Palvelinpuolen web-sovelluskehykset

Palvelinpuolen web-sovelluskehykset ovat tavallisesti asiakaspuolen web-sovelluskehyksiä isompia, koska ne tarvitsevat enemmän eri ominaisuuksia. Palvelimella on tavallisesti yhteys tietokantaan, joka jo sinällään asettaa haasteita muun muassa tietoturvan suhteen. Palvelimen täytyy myös pystyä palvelemaan useaa käyttäjää samanaikaisesti.

Palvelinpuolen web-sovelluskehykset sisältävät yleensä vähintään seuraavat ominaisuudet sisäänrakennettuina:

- tuki HTTP-pyyntöjen vastaanottamiseen ja lähettämiseen
- tuki REST-rajapintojen tekoon
- käyttäjätilit
- lomakkeet ja niiden validointi
- välimuisti ja istunnot
- sivupohjamoottori
- tuki monelle eri tietokannalle
- ORM-järjestelmä
- MVC-arkkitehtuuri tai vastaava
- testaustyökalut
- suojaus yleisiä haavoittuvuuksia vastaan

(Niemi 2015.)

3.4.1 ASP.NET Core

ASP.NET Core on Microsoftin kehittämä ja markkinoima web-sovelluskehys. Se on ASP.NET-sovelluskehysten uusin ja modernein iteraatio ja se julkaistiin vuonna 2016. ASP.NET Core on avointa lähdekoodia ja toi mukanaan huomattavasti aiempaa paremman tuen eri käyttöjärjestelmille. Microsoftin lisäksi myös käyttäjät pääsevät osallistumaan tuotteen jatkokehitykseen. ASP.NET Core on rakennettu Common Language Runtimen päälle, joten se tukee kaikkia .NET kieliä, joista yleisimmät ovat C# ja Visual Basic .NET. (Microsoft 2018.)

3.4.2 Express

Express on Node.js-pohjainen mikrosovelluskehys. Node.js-pohjan ansiosta se toimii asynkronisesti, joka tekee siitä toiminnaltaan varsin erilaisen muihin tässä työssä mainittuihin web-sovelluskehyksiin verrattuna. Express on projektin arkkitehtuurin kannalta hyvin avoin, eikä tee juurikaan päätöksiä ohjelmoijan puolesta.

Vaikka Express on suosituin Node.js-pohjainen sovelluskehys, se on sisäänrakennetuilta ominaisuuksiltaan esimerkiksi Laraveliin verrattuna hyvin minimaalinen ja vastaa lähinnä Flask-sovelluskehystä. Expressillä on rakennettu esimerkiksi tunnetut Paypal.com ja Flickr.com (Wappalyzer n.d.).

3.4.3 Flask

Flask on Armin Ronacherin kehittämä Python-ohjelmointikieltä käyttävä mikrosovelluskehys. Sen kehitys alkoi alun perin aprillipilana, mutta sen yksinkertaisuuden ansiosta saaman suosion myötä sitä alettiin kehittämään tosissaan. Flaskin uusin versio on 1.0. Flaskista löytyy oletuksena muun muassa seuraavat ominaisuudet:

- tuki Jinja2-sivupohjille
- tuki staattisille tiedostoille
- URL-reititys
- istunnot
- yksinkertainen testipalvelin

Alla oleva muutamaan riviin mahtuva esimerkki tulostaa ”Hei Maailma!”:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hei Maailma!"
```

3.4.4 Laravel

Laravel on vuonna 2011 julkaistu ilmainen avointa lähdekoodia oleva PHP-pohjainen web-sovelluskehys. Myös Laravel noudattaa MVC-arkkitehtuuria. Laravelistä löytyy suora tuki Vue-komponenteille, joita sen sivupohjamoottorikin tukee.

Laravelin kehitys alkoi, kun Taylor Otwell halusi luoda paremman vaihtoehdon PHP-pohjaiselle CodeIgniter-sovelluskehykselle. Ensimmäinen versio julkaistiin vuonna 2011 ja se sisälsi jo kehittyneitä ominaisuuksia, kuten lokalisoinnin ja käyttäjien hallinnan. (O’Brien 2016.)

Suhteellisen nuoresta iästään huolimatta se on yksi suosituimmista PHP-pohjaisista web-sovelluskehyksistä. Erityisen suosittu se on uusien kehittäjien keskuudessa. (ValueCoders 2018.)

3.4.5 Ruby on Rails

Ruby on Rails on vuonna 2004 julkaistu Ruby-ohjelmointikieltä käyttävä avointa lähdekoodia oleva web-sovelluskehys. Se käyttää useiden muiden

web-sovelluskehysten tapaan suosittua MVC-arkkitehtuuria. Ruby on Rails syntyi osana Basecamp-nimistä projektinhallintatyökalua, josta kyseisen työkalun kehittäjä David Heinemeier Hansson julkaisi sen erillään. Vuodesta 2007 lähtien Ruby on Rails on tullut Applen Mac OS -käyttöjärjestelmän mukana, mikä auttoi sen nousemista yhdeksi suosituimmaksi web-sovelluskehykseksi. (Basu 2013.)

4 PYTHON-OHJELMOINTIKIELI

Python on vuonna 1990 julkaistu tulkattava korkeantason ohjelmointikieli. Yleisesti Pythonia pidetään yhtenä helpoimmista ja selkeimmistä ohjelmointikielistä helpon luettavuuden takia. Web- ja työpöytäsovellusten lisäksi Python on erittäin suosittu tieteellisessä laskennassa ja sille on kehitetty useita laskennassa käytettäviä kirjastoja. Python-ohjelmointikielen etuna on myös tuki Python Package Indexissä oleville avoimen lähdekoodin paketeille. Huhtikuussa 2018 eri paketteja oli jaossa yli 136 000.

Pythonista löytyy kaksi eri versiota, joista kumpaakin päivitetään erikseen. Pythonin versio 3 julkaistiin joulukuussa 2008 ja se toi mukanaan monia uudistuksia. Uudistusten ja muutosten takia versio 3 ei ole taaksepäin yhteensopiva version 2 kanssa. Python 2 on vielä laajasti käytössä, koska läheskään kaikista kolmansien osapuolten paketeista ei löydy vielä versiota uudemmalle Python 3 -versiolle. Myös monet hieman vanhemmat sovellukset pysyvät tarkoituksella Python 2 -versiossa, koska refaktorointi ei toisi merkittävää lisäarvoa. (Google for Education 2017a.)

4.1 Pythonin suosio

Python on erittäin suosittu ohjelmointikieli. GitHubin julkaiseman suosituimmat ohjelmointikielet -listan mukaan Python oli toisiksi suosituin (Github.com 2017.). IEEE Spectrum -lehden tilastojen mukaan Python oli kaikista suosituin heinäkuussa 2017 (IEEE Spectrum 2017.). Huhtikuussa 2018 Python oli TIOBE indeksin mukaan neljänneksi suosituin ohjelmointikieli (Kuva 2).

Apr 2018	Apr 2017	Change	Programming Language	Ratings	Change
1	1		Java	15.777%	+0.21%
2	2		C	13.589%	+6.62%
3	3		C++	7.218%	+2.66%
4	5	^	Python	5.803%	+2.35%
5	4	v	C#	5.265%	+1.69%
6	7	^	Visual Basic .NET	4.947%	+1.70%
7	6	v	PHP	4.218%	+0.84%
8	8		JavaScript	3.492%	+0.64%
9	-	^^	SQL	2.650%	+2.65%
10	11	^	Ruby	2.018%	-0.29%

Kuva 2. 10 suosituinta ohjelmointikieltä TIOBE indeksin mukaan.

4.2 Tulkattavuus

Python on tulkattava ohjelmointikieli, eli sitä ei tarvitse erikseen kääntää suoritusta varten, toisin kuin esimerkiksi C#. Tulkattavuudesta johtuen ohjelmien kehittäminen on nopeaa, mutta suorituskyky ei ole käännettävien kielten tasolla. Kieli on mahdollista kääntää tavukoodiksi, jonka suorittaminen on huomattavasti nopeampaa.

Yksi tulkattavien kielten huonoista puolista on Pythonin tapa tarkistaa koodi vasta, kun ohjelma on suorittamassa kyseessä olevaa koodiriviä. Tästä johtuen esimerkiksi yksinkertaiset kirjoitusvirheet tulevat ilmi vasta silloin, kun ohjelmaa suoritetaan virheen sisältävään riviin asti. Isoissa sovelluksissa on mahdollista, että yksinkertaiset virheet jäävät huomaamatta pitkäksikin aikaa, jollei koodia ole hyvin testattu, mieluiten automaattisesti. Esimerkiksi C#-kielen kohdalla virhe huomattaisiin heti kääntövaiheessa. (Google for Education 2017b.)

4.3 Tyypitys

Python on dynaamisesti ja vahvasti tyypitetty ohjelmointikieli. Dynaamisella tyypityksellä muuttujien ja funktioiden palautusarvojen tyyppejä ei tarvitse määrittää, vaan tulkki määrittää ne automaattisesti annetun arvon perusteella. Tyyppi voi myös muuttua kesken ohjelman. Dynaamisen tyypityksen vastakohta on staattinen tyypitys, tällöin muuttujat ovat ohjelman alusta asti tiettyä tyyppiä, eikä tyyppi voi vaihtua ohjelman ajon aikana. (Python Wiki n.d. b.)

Vahva tyypitys tarkoittaa, että muuttujilla voi tehdä vain niitä asioita, joita niiden tyyppi tukee. Esimerkiksi merkkijonoa ja kokonaislukua ei yleensä voi summata, koska merkkijonon ja kokonaisluvun yhteenlaskua ei ole määritetty. Vahvaa ja staattista tyypitystä pidetään usein virheellisesti synonyymeina, vaikka ne tarkoittavat eri asiaa.

(Python Wiki, n.d. b.)

4.4 Syntaksi

Pythonin syntaksi on useimmista muista suosituista kielistä hieman poikkeava. Rivejä ei lopeteta puolipisteeseen, eikä eri lohkoja ympäröidä aaltosulkeilla. Aaltosulkeiden sijaan lohkoihin kuuluvat rivit määritetään sisentämällä. Python on myös erittäin tarkka siitä, miten sisennys on tehty. Esimerkiksi välilyöntien sekoittaminen tabulaattorilla tehtyihin väleihin lasketaan virheeksi, eikä ohjelmaa voida suorittaa. Python-tiedostot käyttävät .py-tiedostopäätettä ja niitä kutsutaan moduuleiksi.

4.4.1 Funktiot

Funktiot määritetään *def*-avainsanalla, jonka jälkeen seuraa funktion nimi, sulkeet ja kaksoispiste. Funktion sisälle kuuluvat rivit sisennetään. Koska Python on dynaamisesti tyypittävä kieli, argumenttien tai palautettavan arvon tyyppiä ei määritetä.

```
def sum(number_one, number_two):
    sum_of_arguments = number_one + number_two
    return sum_of_arguments
```

Funktioita kutsutaan samantyylisesti muiden yleisten ohjelmointikielten kanssa.

```
example_sum = sum(3, 4)
print(example_sum)
```

4.4.2 Silmukat

Silmukkatyyppejä Pythonista löytyy kaksi erilaista, useista muistakin kielistä tutut *for* ja *while*. For-silmukkaa käytetään tavallisesti, kun jokin operaatio halutaan suorittaa tietyn monta kertaa. *While*-silmukkaa käytetään, kun operaatio halutaan suorittaa niin kauan, kunnes silmukalle annettu ehto muuttuu. Funktioiden tapaan silmukkaan kuuluva lohko määritetään sisentämällä. Silmukoita voi myös laittaa sisäkkäin. (Python Wiki, n.d. a)

Seuraava esimerkkinä toimiva for-silmukka tulostaa *Hello!*-merkkijonon jokaisella silmukan ajokerralla. *Range* on Python-funktio, joka palauttaa listan lukuja. Tässä esimerkissä se palauttaa luvut 0-19, eli silmukka ajetaan 20 kertaa.

```
for x in range(0, 20):
    print("Hello!")
```

For-silmukan vaatima sarja voi olla muutakin, kuin pelkkiä lukuja. Esimerkiksi merkkijono käy sellaisenaan. Alla olevassa esimerkissä silmukka käy läpi *Hello!*-merkkijonon kirjain kerrallaan.

```
example_string = "Hello!"
for x in example_string:
    print(x)
```

Seuraava *while*-silmukka ajetaan niin kauan, kunnes *example_bool*-muuttujan arvo muuttuu epätodeksi. Tässä esimerkissä silmukkaa ajettaisiin loputtomiin.

```
example_bool = True
while example_bool:
    print("It is still true")
```

4.4.3 Luokat

Luokan määritys aloitetaan *class*-avainsanalla, jonka jälkeen määritetään luokan nimi, ja sen perään kaksoispiste. Luokan sisällä oleva lohko määritetään funktioiden tapaan sisentämällä. Lohko voi sisältää muun muassa muuttujia, muiden funktioiden kutsuja ja luokkaan kuuluvien funktioiden määrityksiä. Tavallisesti luokasta löytyy vähintään konstruktori, joka on *__init__*-niminen funktio.

```
class ExampleBaseClass:
    def __init__(self, example_value1, example_value2):
        self.example_value1 = example_value1
        self.example_value2 = example_value2
```

Luokat voivat myös periytyä muista luokista. Luokka, josta peritään, määritetään luokan jälkeen tulevien sulkeiden sisälle.

```
class ExampleDerivedClass(ExampleBaseClass):
    def example_function():
        print("This is a derived class")
```

Uusi *ExampleDerivedClass*-objekti luodaan ja *example_function*-funktiota kutsutaan seuraavalla tavalla:

```
x = ExampleClass(y, z)
x.example_function()
```

4.5 PEP 8

PEP 8 on Python-ohjelmointikielen virallinen tyyliopas, joka sisältää ohjeita muun muassa välilyöntien määrään ja funktioiden nimeämiseen. Tyyliopaan pääasiallinen tarkoitus on antaa ohjeet yhtenäisen ja selkeän koodin kirjoittamiseen. Pythonin mukana tuleva standardikirjasto noudattaa PEP 8 -tyyliopasta. (Python.org 2013.)

PEP 8 -tyylioppaan noudattamisen tärkeys korostuu silloin, kun projektissa työskentelee useita henkilöitä. Näin koodi säilyy yhtenäisen näköisenä. Esimerkiksi avoimen lähdekoodin projekteissa saattaa mukana olla satoja eri henkilöitä. Tärkeintä on kuitenkin käyttää samaa tyyliä, kuin projektissa on alun perinkin käytetty, vaikka se eroaisi joiltakin osin PEP 8 -tyylioppaan ohjeista. (Python.org 2013.)

5 DJANGO

Django on ilmainen vuonna 2005 julkaistu avointa lähdekoodia oleva web-sovelluskehys, joka käyttää Python-ohjelmointikieltä. Arkkitehtuurina Django käyttää MVC-arkkitehtuurin kanssa erittäin samankaltaista MVT-arkkitehtuuria. Djangoa käyttäen on kehitetty erittäin suosittuja sivustoja, kuten Pinterest, Instagram ja BitBucket.

Django sisältää monia web-sovelluksissa usein tarvittavia ominaisuuksia. Sovelluskehyksestä löytyy muun muassa erittäin kattava käyttäjätilijärjestelmä, joka tukee eri käyttäjätasoja ja ryhmiä. Käyttäjätilijärjestelmää voi myös laajentaa kolmansien osapuolin paketeilla, joilla pystyy helposti lisäämään muun muassa tuen OAuth-kirjautumiselle. Djangon oma ORM-järjestelmä tekee tietokantojen käytöstä kehittäjälle yksinkertaista ja tehokasta. Sivupohjamoottorina toimii Djangon oma Jinja2-sivupohjamoottoria muistuttava moottori.

Django on hyvin vahvasti muokattavissa. Halutessaan ison osan sisäänrakennetuista ominaisuuksista voi vaihtaa kolmansien osapuolien kehittämiin ominaisuuksiin. Esimerkiksi Djangon oman ORM-järjestelmän tilalla voi käyttää SQLAlchemyä ja sivupohjamoottorin voi vaihtaa suosittuun Jinja2-moottoriin. (Django Girls n.d.)

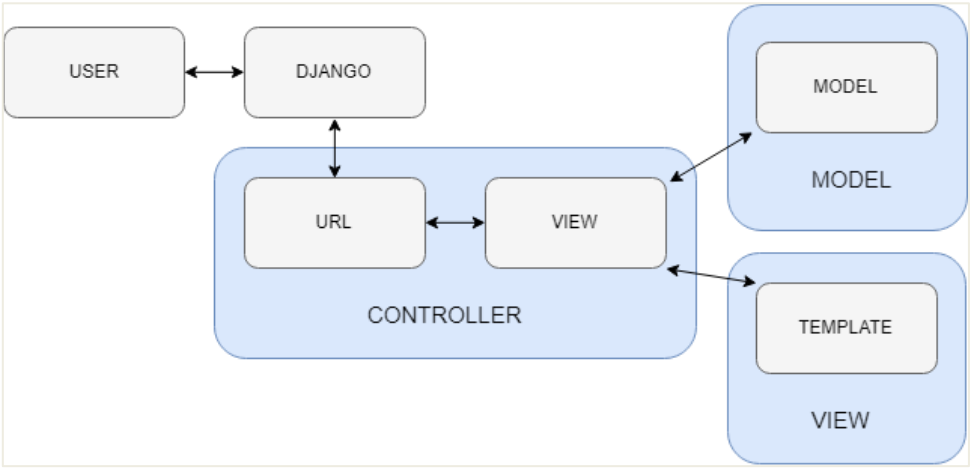
5.1 Historia

Ensimmäinen versio Djangosta syntyi vuonna 2003, kun amerikkalaisen Lawrence Journal-World -lehden ohjelmoijat alkoivat käyttämään Python-ohjelmointikieltä web-sovellusten rakentamiseen. Yleisölle tämä versio julkaistiin BSD-lisenssin alla vuonna 2005. Vuodesta 2008 lähtien Djangon kehityksestä on vastannut voittoa tavoittelematon Django Software Foundation -säätiö. (Django Documentation n.d. c.)

Djangon uusi 2.0 versio julkaistiin joulukuussa 2017. Se sisälsi merkittäviä uudistuksia ja ominaisuuksia. Muun muassa URL-reitityksen syntaksi muutettiin yksinkertaisemmaksi ja tuki Python 2 -versiolle lopetettiin. Kirjoitushetkellä Djangon uusin versio on 2.0.4.

5.2 MVT-arkkitehtuuri

Django noudattaa MVT-arkkitehtuuria, joka on Djangon kehittäjien oma näkemys erittäin yleisesti käytössä olevasta MVC-arkkitehtuurista. Yksi merkittävissä eroista on varsinaisen käsittelijän puuttuminen. Käsittelijän tilalla Djangossa on URL-määritykset ja näkymät (Kuva 3). URL-määritysten perusteella Django tietää mihin näkymään käyttäjä viedään, jonka jälkeen näkymä hakee halutut tiedot malleista ja renderöi käyttäjälle HTML-sivun käyttäen sivupohjia. (TheDjangoBook n.d. a.)



Kuva 3. MVC- ja MVT-arkkitehtuurien yhteneväisyydet.

MVT- ja MVC-arkkitehtuurien etuna on sovelluksen eri osien erottaminen toisistaan. Tämän ansiosta näkymät, mallit ja sivupohjat voidaan tehdä toisistaan erillään, eikä esimerkiksi mallin tarvitse tietää, mitä näkymässä tai sivupohjassa tapahtuu.

5.3 Mallit

Mallit ovat Python-luokkia, jotka kuvaavat tietokantaan tallennettavaa dataa. Mallien avulla tietokannan dataa voi käsitellä oliomuotoisena, eli tavallisten SQL-kyselyiden sijaan dataa voi muokata ja hallita samanlailla, kuin Python-objekteja. Oletuksena Django käyttää sen omaa ORM-järjestelmää, mutta sen voi korvata myös jollakin muulla kolmannen osapuolen ORM-järjestelmällä. (TheDjangoBook n.d. b.)

Mallien määrittämisellä Python-koodina pelkkien tietokantojen sijaan on useita etuja. Datan muuttaminen ja käyttäminen oliomuodossa on yksinkertaista, koska Django tietää jo valmiiksi mallien kentät. Lisäksi sama malli toimii riippumatta käytetystä tietokantatyypistä. Myös versionhallintajärjestelmät ovat paremmin yhteensopivia Python-tiedostojen kanssa. (TheDjangoBook n.d. b.)

5.3.1 Mallien määritys

Mallit määritetään sovelluskohtaiseen models.py-tiedostoon. Mallien perusluokkana tulee käyttää *django.db.models*-kirjastosta löytyvää *Model*-luokkaa. Luokalle määritetyt muuttujat kuvaavat tietokantaan tulevia kenttiä. Seuraava esimerkki määrittää mallin *Person*, joka sisältää kentät etunimelle ja sukunimelle.

```
class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

Määritetty malli luo tietokantaan uuden taulun alla olevan esimerkin mukaisesti. Taulun nimi koostuu sovelluksen ja mallin nimestä. *ID*-kenttä on Djangon automaattisesti lisäämä primääriavaimena toimiva kenttä.

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

5.3.2 Tietokantakyselyiden tekeminen

Tavallisesti SQL-pohjaisten tietokantojen kyselyt tehtäisiin SQL-kieltä käyttäen. Djangon ORM-järjestelmän avulla kyselyiden tekeminen on kehittäjän näkökulmasta huomattavasti yksinkertaisempaa. Esimerkiksi uusi *Person*-tietue tehdään samanlailla, kuin uusi instanssi tavallisesta Python-luokasta.

```
from .models import Person
new_person = Person(
    first_name="Esko",
    last_name="Example"
)
new_person.save()
```

Tietue tallentuu tietokantaan vasta, kun *save*-funktio kutsutaan. Pinnan alla Django käyttää tavallista SQL-kielen *INSERT*-lauseketta tietueen tekemiseen. (Django Documentation n.d. a.)

5.4 Näkymät

Näkymät ovat Python-funktioita, jotka vastaanottavat pyynnön ja palauttavat vastauksen. Palautettava vastaus voi olla esimerkiksi HTML-koodia tai 404-virhe. Näkymässä oleva koodi päättää mitä palautettava vastaus sisältää. Tavallisesti näkymät määritetään sovelluskohtaiseen *views.py*-tiedostoon. Näkymät voivat olla joko funktio- tai luokkapohjaisia.

5.4.1 Funktiopohjaiset näkymät

Funktiopohjaiset näkymät ovat rakenteeltaan vapaita ja yksinkertaisia. Ne määritetään samanlailla, kuin tavalliset Python-funktiot. Pyynnön HTTP-metodia vastaava lohko valitaan ehtolauseketta käyttäen. Funktiopohjaisten näkymien etu on niiden yksinkertaisuus ja selkeys. Kaikki näkymän käyttämä koodi on selkeästi näkyvillä, eikä piilotettuna perusluokassa. (Freitas, 2017)

```
def example_view(request):
    if request.method == "POST":
        # Code block for POST request
    else:
        # Code block for other requests
```

5.4.2 Luokkapohjaiset näkymät

Luokkapohjaiset näkymät määritetään samanlailla, kuin tavalliset Python-luokat. Funktiopohjaisista näkymistä poiketen jokaista käytettävää HTTP-metodia varten määritetään metodin nimeä käyttävä funktio.

Luokkapohjaisten näkymien merkittävin etu on mahdollisuus periyttää niitä muista näkymistä. Djangon ydinkirjasto sisältää monia niin sanottuja *generic*-näkymiä, jotka sisältävät usein tarvittavia ominaisuuksia. Esimerkiksi *FormView*-näkymää laajentamalla voi nopeasti tehdä näkymän lomaketta varten. Yksinkertaisin *generic*-näkymä on View, joka hoitaa muun muassa käytettävää HTTP-metodia vastaavan funktion kutsumisen automaattisesti. (Freitas 2017.)

```
class ExampleView(View):
    def get(self, request):
        # Code block for GET request

    def post(self, request):
        # Code block for POST request
```

5.5 URL-määritys

URL-määritysten avulla käyttäjän lähettämä pyyntö ohjataan oikeaan näkymään. Djangossa URL-määritys tapahtuu määrittämällä eri reiteille takaisinkutsufunktiot, joita kutsutaan, kun pyyntö tulee kyseiseen osoitteeseen. Perinteisesti määritykset tallennetaan *urls.py*-tiedostossa olevaan *urlpatterns*-listaan.

Path-funktio vastaanottaa ensimmäisenä parametrina lausekkeen osoitteelle ja toisena näkymän, johon käyttäjä halutaan ohjata. Alla olevassa esimerkissä *path*-funktio ohjaa pyynnön *special_case_2003*-näkymään, kun osoite on muotoa *domain/articles/2003/.*

```
urlpatterns = [
    path("articles/2003/", views.special_case_2003),
```

```
]
```

Osoitteesta voi myös ottaa arvoja talteen ympäröimällä haluttu kohta kulmasulkeilla. Samalla voi myös määrittää arvon tyypin. Mikä tahansa merkkijono kelpaa, jos tyyppiä ei erikseen määritetä. Alla oleva esimerkki lähettää *year*-kohdalla olevan luvun *year*-nimisenä avainsana-argumenttina näkymälle.

```
path("articles/<int:year>/", views.year_archive)
```

Näkymässä osoitteesta talteen otettua osaa pystyy käyttämään seuraavalla tavalla:

```
wanted_year = kwargs["year"]
```

Path-funktion sijasta voi käyttää säännöllisiä lausekkeita tukevaa *re_path*-funktiota. Syntaksi on muuten samanlainen, mutta ensimmäisenä parametrina annetaan säännöllinen lauseke. Tämä soveltuu *path*-funktiota paremmin monimutkaisiin URL-määrityksiin.

```
re_path(r"^articles/(?P<year>[0-9]{4})/$", views.year_archive)
```

5.6 Sivupohjat

Sivupohjat ovat tekstitiedostoja, jotka sisältävät esimerkiksi HTML-elementtejä. Sivupohjia ei ole ainoastaan rajattu HTML-muotoon, vaan niitä voi käyttää missä tahansa tekstipohjaisessa tiedostomuodossa (HTML, XML, CSV, jne.). Djangossa ei ole pakko käyttää Djangon omaa sivupohjamoottoria, vaan sen voi korvata esimerkiksi Jinja2-moottorilla. (Django Documentation n.d. b.)

Sivupohjien merkittävin hyöty staattisten HTML-tiedostojen sijaan on mahdollisuus dynaamiselle datalle, joka voi esimerkiksi riippua sivustolle tulleesta käyttäjästä. Sivupohjat voivat myös laajentaa toisiaan. Tavallisesti sivuston yleinen ulkoasu tehdään *base.html*-sivupohjaan, jota myöhemmin laajennetaan. Näin sivuston tekijä säästyy jatkuvalta HTML-elementtien uudelleenkirjoitukselta ja voi keskittyä vaan vaihtuvaan sisältöön. (Django Documentation n.d. b.)

5.6.1 Sivupohjien syntaksi

Sivupohja voi tarvittaessa sisältää muuttujia, jotka korvataan arvoilla, kun sivupohjaa käyttävä sivu ladataan. Muuttujat ympäröidään kaksinkertaisilla aaltosulkeilla.

```
{{ user.first_name }}
```

Muuttujien näyttämää arvoa voi myös muokata käyttämällä suodattimia. Django sisältää monia valmiita suodattimia, mutta niitä voi myös tehdä itse. Alla oleva esimerkki näyttää *name*-muuttujan arvon pienaakkosina.

```
{{ name|lower }}
```

Muuttujien lisäksi tarjolla on tageja, joiden avulla voi esimerkiksi tehdä *for*-silmukan sivupohjan sisältöön. Tagit ympäröidään aaltosulkujen ja prosenttimerkkien yhdistelmällä.

```
{% extends "base.html" %}
```

Usein sivupohjat sisältävät yhdistelmän muuttujia ja tageja. Alla oleva esimerkki tulostaa jokaisen *users*-listassa olevan objektin etunimen h1-tason otsikkona pienaakkosina.

```
{% for user in users %}
<h1>{{ user.first_name|lower }}</h1>
{% endfor %}
```

5.6.2 Periytyminen

Sivupohjat pystyvät laajentamaan toisiaan, jolloin vältytään HTML-elementtien uudelleenkirjoittamiselta. Sivupohjaan on mahdollista tehdä lohkoja, joiden sisällön voi myöhemmin korvata toisessa sivupohjassa olevalla samannimisellä lohkolla. Tavallisesti *base.html*-sivupohja pitää sisällään sivuston rakenteen ja ainoastaan sisältöä vaihdetaan käyttäen lohkoja. *Extends*-tagilla määritetään sivupohja, josta halutaan periytyä.

Alla olevassa esimerkissä luodaan *content*-lohko.

```
<!-- base.html -->
<!doctype html>
<html>
    <head>
        <title>Block example</title>
    </head>
    <body>
        {% block content %}
        <h1>This is the original content</h1>
        {% endblock content %}
    </body>
</html>
```

Lohkon sisällön voi kokonaan vaihtaa luomalla uuden sivupohja-tiedoston, joka periytyy sivupohjasta, jossa korvattava lohko on. Alla oleva esimerkki korvaa *content*-lohkon sisällön eri otsikolla, mutta muuten sivupohja pysyy täysin samana.

```
<!-- different_content.html -->
{% extends "base.html" %}

{% block content %}
```

```
<h1>This is the new content</h1>
{% endblock content %}
```

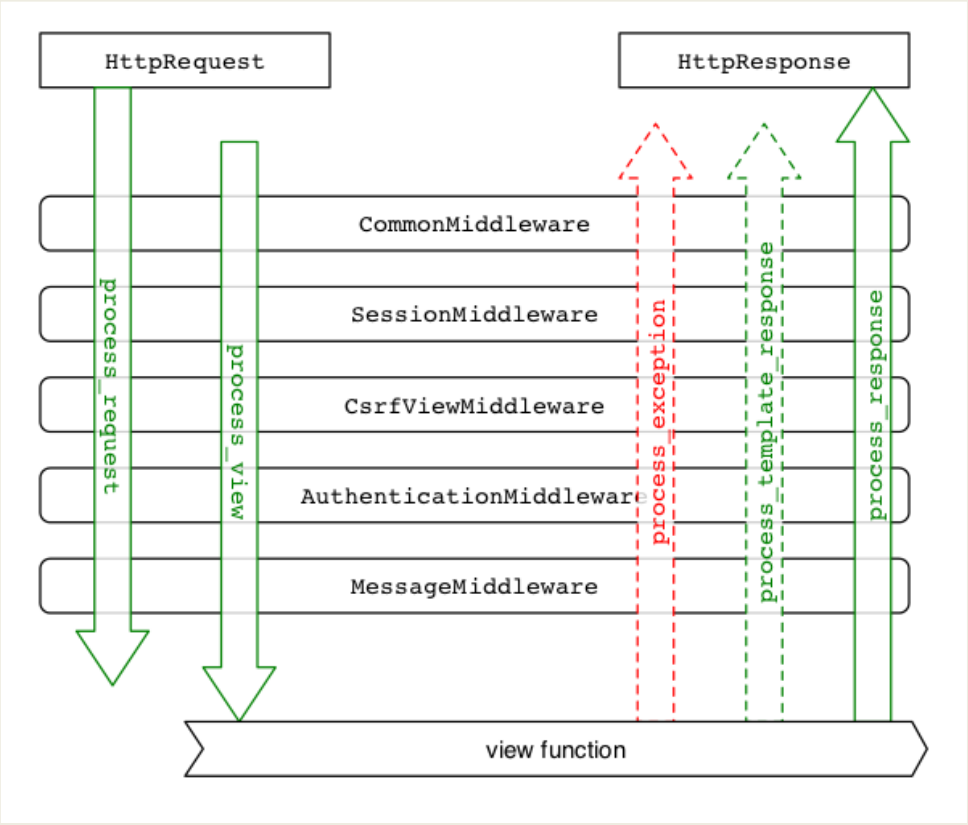
5.6.3 Include

Include-tagilla voidaan sisällyttää toinen sivupohja nykyiseen sivupohjaan. Tätä käytetään usein esimerkiksi navigaation tai alatunnisteen lisäämiseen. Hyötynä tästä on, että navigaatiota tai alatunnistetta voi muokata erikseen ilman, että kaikki on samassa massiivisessa sivupohja-tiedostossa.

```
{% include "main_navigation.html" %}
```

5.7 Middleware

Middleware-luokat toimivat palvelimen saaman pyynnön ja palvelimen lähettämän vastauksen välissä samalla muuttaen request- tai response-objektia (Kuva 4). Middlewarejen hyöty on niiden mahdollistama *request*- tai *response*-objektin muokkaaminen ennen, kuin haluttu näkymä suoritetaan.



Kuva 4. Middleware-luokkien toimintatapa.

Middlewaret pystyvät kiinnittymään niin sanottuihin *hook*-funktioihin, joita kutsutaan automaattisesti, kun pyyntö tai vastaus kulkee käyttäjän ja näkymän välissä. *Hook*-funktioita on viisi erilaista:

- process_request
- process_view

- process_response
- process_template_response
- process_exception

(Akshar, 2015)

Middlewaret aktivoidaan lisäämällä ne *settings.py*-tiedostossa olevaan *MIDDLEWARE*-listaan. Middlewarejen järjestyksellä on väliä, koska ne voivat olla riippuvaisia muista middlewareista. Esimerkiksi oletuksena aktiivisena oleva *AuthenticationMiddleware* käyttää *SessionMiddlewaren* hallitsemaa istuntoa.

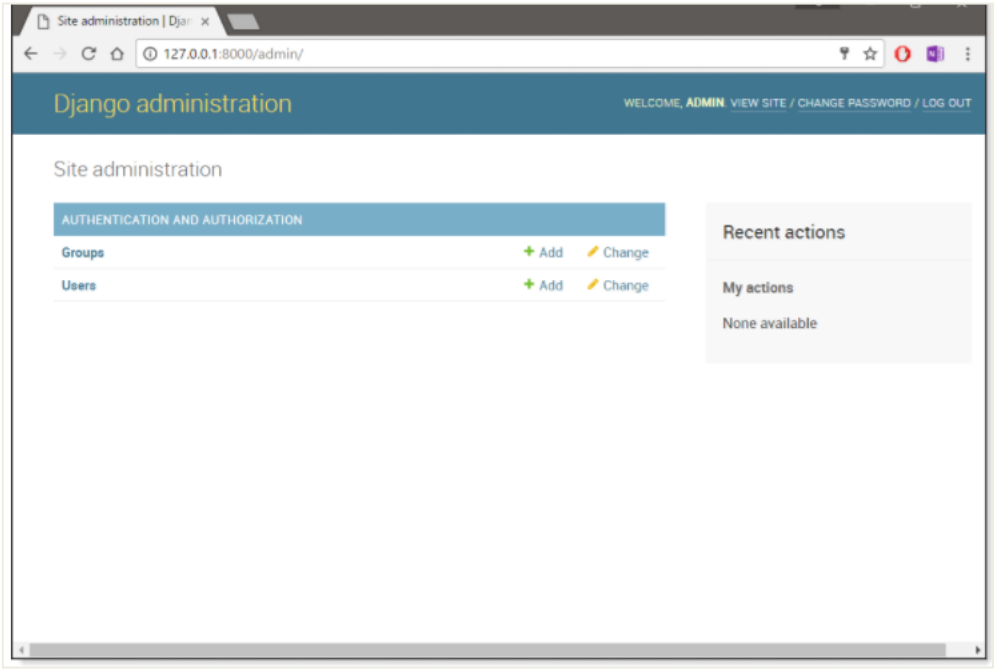
```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Seuraava middleware-esimerkki tallentaa käyttäjän *Profile*-mallissa määritetyn aikavyöhykkeen istuntodataan, jolloin se on helposti käytettävissä kaikissa näkymissä.

```
class TimezoneMiddleware(object):
    def process_request(self, request):
        request.session["timezone"] = request.user.profile.timezone
```

5.8 Admin-sivusto

Djangosta löytyy valmiina automaattisesti luotu ylläpitäjälle tarkoitettu sivusto, jonka avulla voi tarkastella ja muokata tietokannassa olevaa dataa. Sivustoa ei ole tarkoitettu sovelluksen tavallisille käyttäjille, vaikka se siihen soveltuisikin muutaman muokkauksen jälkeen. Sivustolla näkyy oletuksena ainoastaan käyttäjäryhmät ja käyttäjät, koska ne ovat valmiiksi rekisteröity admin-sivustoa varten (Kuva 5).



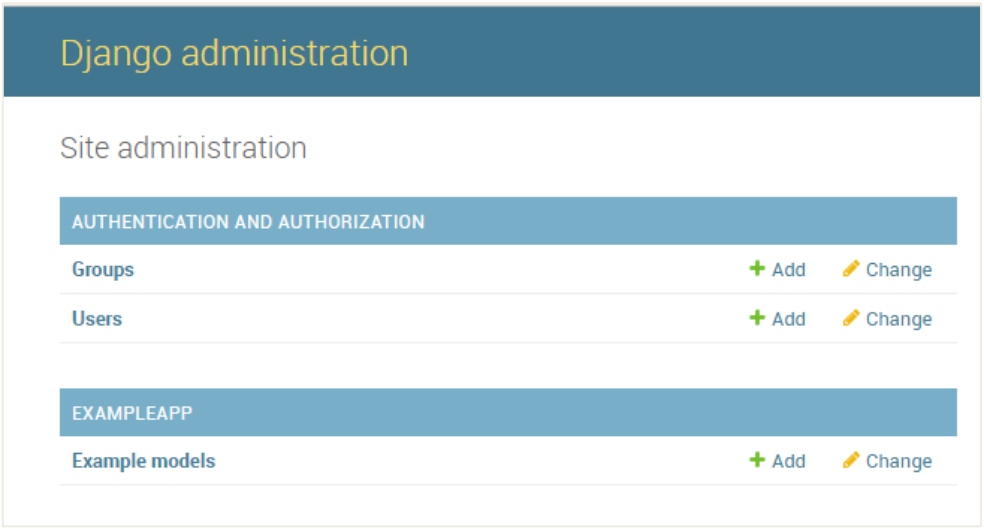
Kuva 5. Admin-sivuston oletusnäkymä.

Mallit täytyy erikseen rekisteröidä admin-paneelia varten ennen, kuin ne näkyvät siellä. Malli rekisteröidään *django.contrib.admin.site*-kirjastossa olevalla *register*-funktiolla, jonka parametriksi annetaan haluttu malli. Tavallisesti mallit rekisteröidään sovelluskohtaisessa *admin.py*-tiedostossa.

```
from django.contrib import admin
.models import ExampleModel

admin.site.register(ExampleModel)
```

Rekisteröinnin jälkeen myös *ExampleModel*-mallin objekteja voi tarkastella ja muokata admin-sivustolla (Kuva 6).

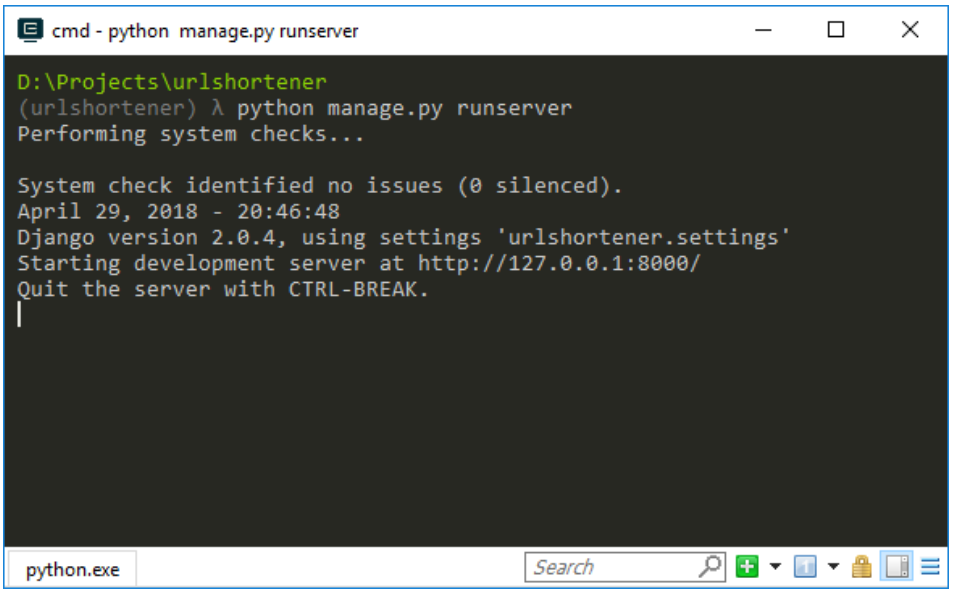


Kuva 6. ExampleModel-malli lisättynä admin-sivustolle.

6 PROJEKTISSA KÄYTETTÄVÄT TYÖVÄLINEET JA OHJELMAT

6.1 Cmder

Cmder on Samuel Vaskon kehittämä ConEmu-pohjainen konsoliemulaattori Windowsille. Cmderin etuina tavalliseen komentoriviin on muun muassa valmis tuki Linux- ja Git Bash -komennoille. Cmder tukee myös välilehtiä, mikä helpottaa esimerkiksi Djangon testipalvelimen ja versionhallinnan samanaikaista käyttöä. Työkalun ulkoasu on monipuolisesti muokattavissa. Oletuksena se käyttää tekstieditoreistakin tuttua Monokai-teemaa (Kuva 7).



Kuva 7. Cmder-työkalun oletuksena käyttämä Monokai-teema.

6.2 Git ja GitHub

Git on Linus Torvaldsin kehittämä hajautettu versionhallintajärjestelmä, jonka ensimmäinen julkinen versio julkaistiin vuonna 2005. Se on ilmainen ja avointa lähdekoodia, ja se on tällä hetkellä maailman eniten käytetyin versionhallintajärjestelmä. Hajautetulla versionhallintajärjestelmällä tarkoitetaan sitä, että palvelimen lisäksi myös jokaisella asiakaskoneella on kopio koko tietolähteestä. Hyötynä tästä on se, että vaikka palvelin hajoaisi, tiedot pysyvät silti asiakaskoneilla. (Atlassian n.d.)

GitHub on palvelu, joka tarjoaa tilaa Git-versionhallintajärjestelmää käyttäville ohjelmistoprojekteille. Tilan lisäksi GitHub tarjoaa erilaisia Git-projektien hallintaan liittyviä ominaisuuksia, kuten tehtävien ja bugien seurannan. GitHubin peruskäyttö on ilmaista, mutta esimerkiksi yksityiset repositoryt ovat maksullisia. Repository-termillä tarkoitetaan projektin ”säilytyspaikkaa”. (Brown 2016.)

6.3 Virtualenv ja virtualenvwrapper

Virtualenv on työkalu, jonka avulla voi luoda muusta käyttöjärjestelmästä eristetyn ympäristön Python-paketteja varten. Näin samalla tietokoneella voi olla samaan aikaan monta eri versiota esimerkiksi Djangosta. Tämä on tarpeellista, jos työstää samalla koneella useaa eri projektia, jotka vaativat eri versioita Python-paketeista.

Virtualenvwrapper on paketti, joka sisältää laajennuksia ja uusia ominaisuuksia virtualenv-työkaluun. Sen pääasiallinen tarkoitus on helpottaa ja nopeuttaa virtualenv-työkalun käyttöä muun muassa yksinkertaistamalla komentoja.

6.4 Visual Studio Code

Visual Studio Code on Microsoftin tekemä avoimen lähdekoodin tekstieditori, joka luotiin kevyemmäksi vaihtoehdoksi Microsoftin isommalle Visual Studio -ohjelmistokehitysympäristölle. Visual Studio Code käyttää Electron-sovelluskehystä, jonka avulla tehdään Node.js-pohjaisia sovelluksia työpöytäkäyttöön. Visual Studio Code tukee kaikkia kolmea isoa käyttöjärjestelmää, eli Windowsia, macOS:ää ja Linuxia. (Visual Studio Code Documentation n.d.)

7 PROJEKTIN TOTEUTUS

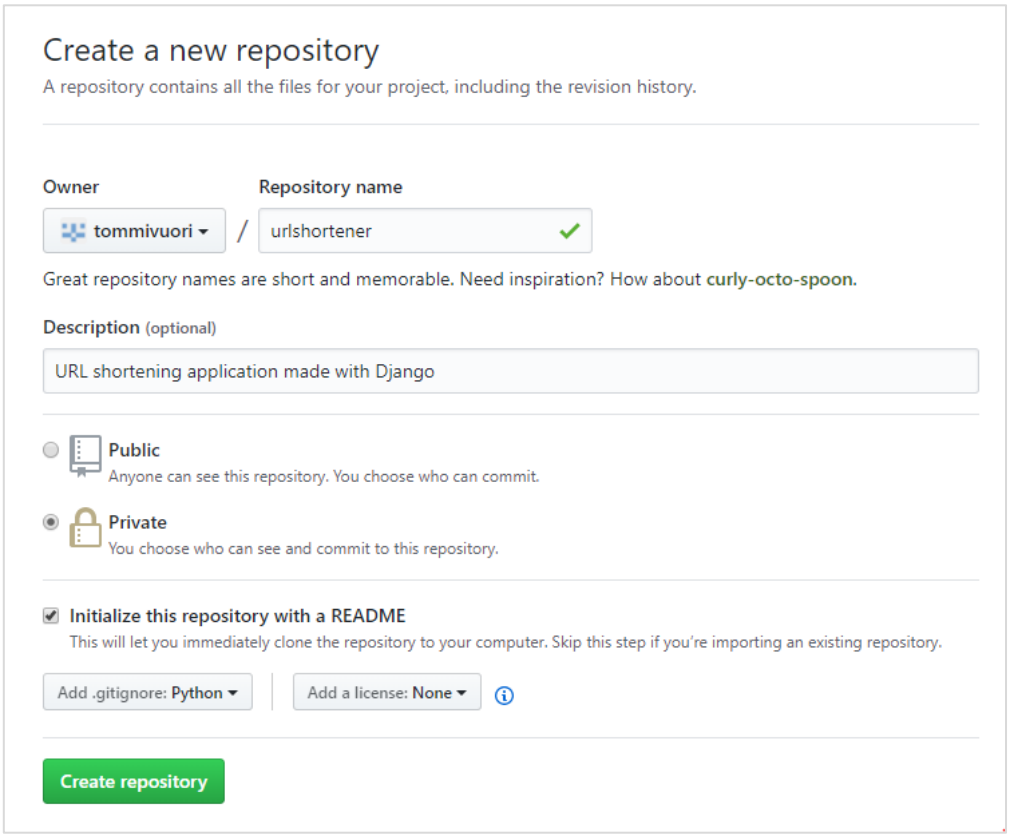
Opinnäytetyön käytännön osuutena loin projektin, jonka tavoitteena oli toteuttaa toimiva web-sovellus Djangolla. Tarkoituksena oli luoda linkin lyhennys -palvelu, johon kuuluisi käyttäjätilit ja omien linkkien hallintaan tehty hallintapaneeli. Inspiraatiota projektin ominaisuuksiin otin Bitly.com-palvelusta, joka on Twitterinkin käyttämä linkkien lyhentämiseen keskittynyt palvelu.

Varsinainen linkin lyhennys -algoritmi oli hyvin yksinkertainen. Generoin satunnaisesti 8-merkkisen merkkijonon, joka sai sisältää isoja ja pieniä kirjaimia sekä numeroita. Jos generoitu merkkijono oli jo käytössä, sovellus generoi uuden merkkijonon, kunnes se oli uniikki. Mahdollisia eri yhdistelmiä on olemassa niin monta, että saman yhdistelmän generointi pitäisi olla erittäin harvinaista, joten en nähnyt tarpeelliseksi paremman algoritmin luontia.

Projektin visuaalisen ilmeen tekemiseen käytin Bootstrap-kirjastoa, joka sisältää ison määrän valmiiksi tehtyjä CSS-tyylimäärittelyjä. Näin minun ei tarvinnut käyttää juurikaan aikaa sivuston visuaaliseen tyyliin ja pystyin keskittymään enemmän Djangoon liittyviin asioihin.

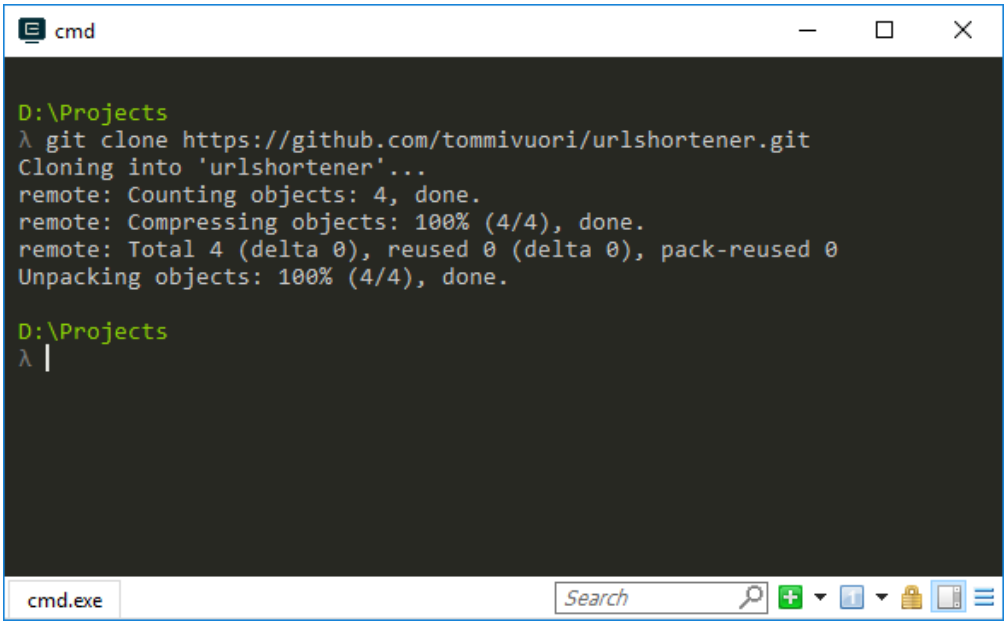
7.1 Gitin käyttöönotto

Aloitin projektin tekemällä GitHubiin uuden repositoryn projektiani varten (Kuva 8). Opiskelijalisenssin ansiosta pystyin tekemään ilmaiseksi myös yksityisiä repositoryjä. Lisäsin *.gitignore*-tiedostoksi GitHubista valmiiksi löytyvän Python-projekteille tarkoitetun *.gitignore*-tiedoston. Tiedosto määritti, mitä tiedostotyyppejä tai kansioita ei lisätty repositoryyn. Tavallisesti nämä olivat automaattisesti generoituvia tiedostoja tai paketteja, jotka asennettiin erikseen esimerkiksi Pip-työkalulla.



Kuva 8. Uuden GitHub-repositoryn luonti.

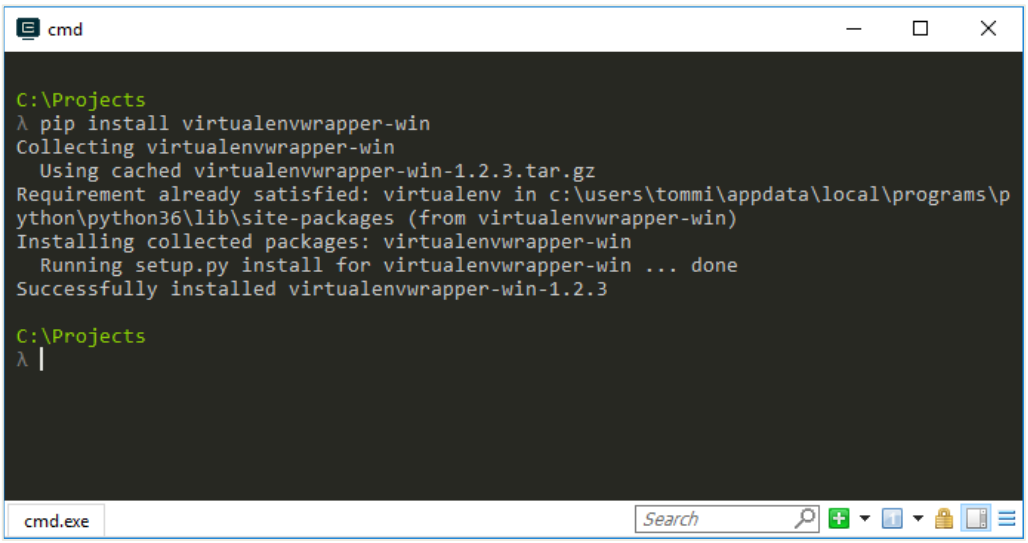
Repository piti seuraavaksi kloonata tietokoneelleni komennolla *git clone* (Kuva 9). Komento kopioi palvelimella olevat tiedostot ja määritti joitakin palvelimeen liittyviä asetuksia valmiiksi.



Kuva 9. Repositoryn kloonaus komennolla git clone.

7.2 Virtualenvwrapperin käyttöönotto

Asensin Virtualenvwrapper-työkalun käyttäen Pythonin mukana tullutta Pip-työkalua, joka oli tarkoitettu Python Package Indexissä olevien pakettien asentamiseen (Kuva 10).



Kuva 10. Virtualenv-asennus käyttäen Pip-työkalua.

Seuraavaksi loin projektiani varten uuden *virtualenv*-ympäristön. Annoin ympäristölle nimeksi *urlshortener*, jota käytin repositoryn ja myöhemmin myös Django-projektin nimenä.

```
mkvirtualenv urlshortener
```

Luotu *virtualenv*-ympäristö aktivoitui automaattisesti. Kun ympäristö oli aktiivisena, kaikki Python-paketit, jotka sovellusta varten tulin asentamaan, asentuivat koko käyttöjärjestelmän sijasta vain aktiivisena olevaan *virtualenv*-ympäristöön.

7.3 Djangon asennus ja uuden projektin luominen

Asensin Djangon uusimman version, joka oli kirjoitushetkellä 2.0.4. Pip-työkalu asensi automaattisesti paketin uusimman version, jollei versiota erikseen määritetty.

```
pip install django
```

Seuraavaksi loin uuden Django-projektin käyttäen *django-admin*-komentorivityökalua, joka sisältyi Djangon asennukseen. Piste komennon lopussa määritti, että projektin tiedostot luotiin nykyiseen kansioon uuden kansion sijasta.

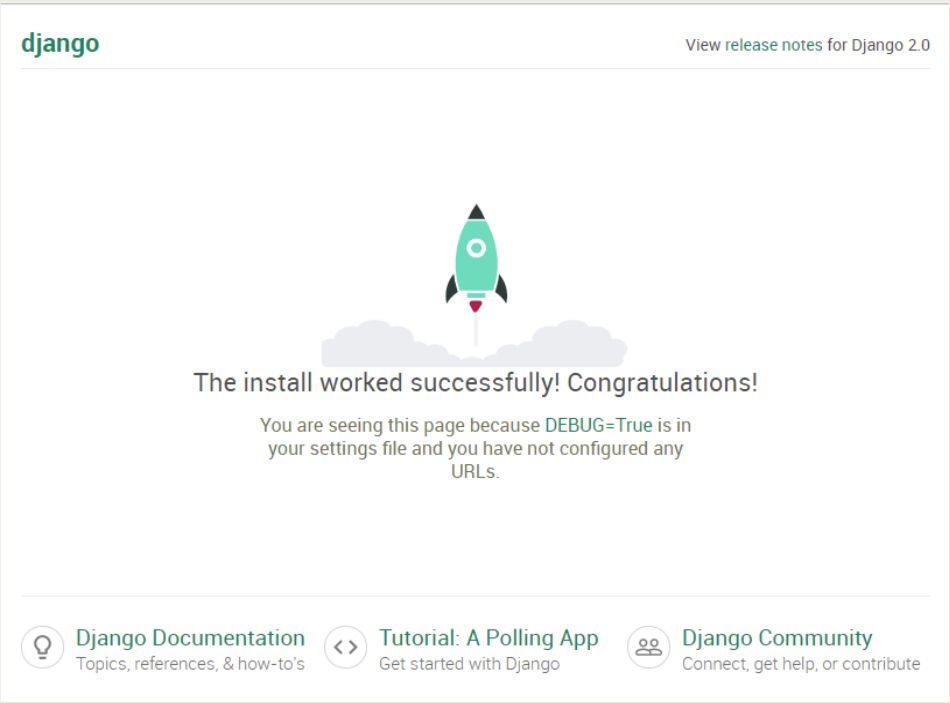
```
django-admin startproject urlshortener .
```

Django loi automaattisesti yksinkertaisen kansiorakenteen projektille.

```
urlshortener/
        manage.py
        urlshortener/
            __init__.py
            settings.py
            urls.py
            wsgi.py
```

Projektin juuressa ollut manage.py oli komentorivityökalu, jolla suoritettiin erilaisia komentoja. Sisempi *urlshortener*-kansio oli Python-paketti ja sen sisällä ollut *__init__.py*-tiedosto oli tyhjä Python-tiedosto, jonka perusteella Python tiesi, että tätä kansiota tuli kohdella Python-pakettina. *Settings.py* oli tiedosto, joka määritti Django-projektiin liittyvät asetukset, kuten sivupohjat sisältävät kansiot tai staattisten tiedostojen sijainnin. *Urls.py* määritti koko projektin URL-osoitteet. Sovelluksille kuului yleensä vielä erilliset omat *urls.py*-tiedostot, jotka tuotiin tähän *include*-funktiota apuna käyttäen.

Asennuksen onnistumisen pystyi nopeasti testaamaan käynnistämällä Djangon mukana tulleen kehityskäyttöön tarkoitetun web-palvelimen. Sivun näkymisestä osoitteessa localhost:8000 tiesi, että asennus oli onnistunut (Kuva 11).



Kuva 11. Djangon oletussivu, josta näki, että asennus onnistui.

7.4 Templates- ja static-kansioiden luonti

Ennen varsinaisen ohjelmoinnin aloitusta loin valmiiksi kansiot projektin sivupohjia ja staattisia tiedostoja varten, koska niitä joka tapauksessa tarvitaan myöhemmin. Staattisilla tiedostoilla tarkoitetaan esimerkiksi CSS- tai JavaScript-tiedostoja. *Templates*-kansio tuli sisältämään sovelluksen käyttämiä HTML-sivupohjia. Loin kummatkin kansiot projektin juureen, eli samaan kansioon kuin missä *manage.py*-työkalu on. Tein *static*-kansioon myös kolme alikansiota, *css*, *images* ja *js*.

Kansioiden luonnin jälkeen ne täytyi vielä määrittää *settings.py*-tiedostossa. Oletuksena Django etsii sivupohjia vain projektiin luotujen sovellusten sisältä. Mielestäni oli kuitenkin selkeämpää pitää esimerkiksi sivuston *base.html*-sivupohja suoraan projektin juuressa.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            os.path.join(BASE_DIR, 'templates'), # add root folder
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
```

```
]

STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'static'),
]
```

Os.path.join on Python-funktio, joka yhdistää tiedostopolkuja. Funktion käyttäminen polkujen suoraan kirjoittamisen sijaan oli suositeltavaa, koska muuten polku voisi olla eri riippuen käyttöjärjestelmästä.

7.5 Base.html-sivupohja

Aloitin projektin luomalla koko sivuston pohjalla toimivan base.html-sivupohjan. Tein sivupohjan projektin juuressa olevaan *templates*-kansioon. Sivupohja sisälsi muun muassa linkit CSS- ja JavaScript-tiedostoihin. Lisäsin sivuston navigaation ja alatunnisteen käyttäen *include*-tagia, jotta niitä pystyisi muokkaamaan kätevästi erikseen. *Body*-elementin sisälle loin lohkon nimeltä *content*, joka tulisi sisältämään tätä laajentavien sivupohjien sisällön.

```
<!-- base.html -->
{% load static %}
<!doctype html>
<html lang="en">
    <head>
        <!-- meta tags, title, bootstrap and custom css -->
    </head>
    <body class="text-center">
        <div class="d-flex min-height-100 flex-column">
            {% include "main_navigation.html" %}
            <main role="main">
                {% block content %}
                {% endblock content %}
            </main>
            {% include "main_footer.html" %}
        </div>
        <!-- bootstrap javascript -->
    </body>
</html>
```

7.6 Accounts-sovellus

Tein projektiin uuden sovelluksen nimellä *accounts*. Tämän sovelluksen oli tarkoitus sisältää kaikki käyttäjätileihin ja niiden hallintaan tarvittavat mallit, näkymät, lomakkeet ja URL-reitit. Uusi sovellus luotiin alla olevalla komennolla.

```
python manage.py startapp shortener
```

Sovellus piti vielä aktivoida. Aktivointi tapahtui lisäämällä se *settings.py*-tiedostossa olevaan *INSTALLED_APPS*-listaan.

```
# settings.py
INSTALLED_APPS = [
```

```
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'accounts',
]
```

7.7 Auth-sovelluksen käyttöönotto

Hyödynsin käyttäjätilien teossa Djangon sisäänrakennettua *auth*-sovellusta, joka sisälsi *User*-mallin ja muutamia valmiita näkymiä, kuten esimerkiksi sisäänkirjautumiseen tarvittavan näkymän. Otin käyttäjätilit käyttöön lisäämällä *auth*-sovelluksen valmiit URL-määritykset projektin urls.py-tiedostoon *include*-funktiolla. Määritin ne alkamaan *accounts/*-merkkijonolla, jolloin esimerkiksi kirjautumiseen käytetty osoite olisi *domain/accounts/login/.*

```
# urls.py
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('django.contrib.auth.urls')),
]
```

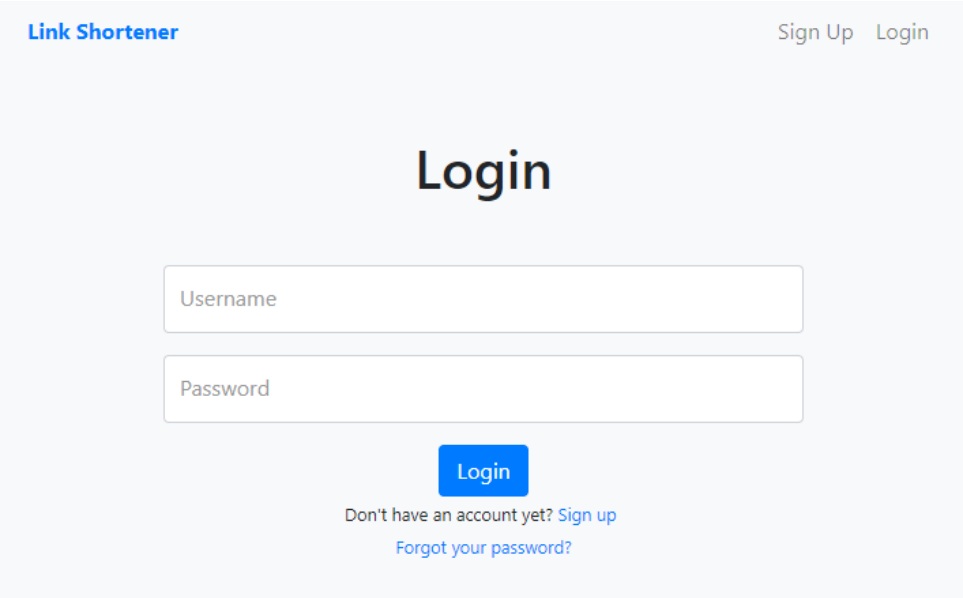
Käyttäjätilit olivat toiminnassa, mutta sivupohjia ei ollut vielä olemassa ja käyttäjätilin tekoon tarvittava näkymä piti tehdä itse. Tein sivupohjat *accounts*-sovelluksen juuressa olevaan *templates*-kansioon. Sivupohjille täytyi tehdä *registration*-niminen alakansio, jotta Django löytäisi ne.

7.8 Sivupohjat auth-sovelluksen näkymille

Seuraavaksi tein sivupohjat jokaiselle *auth*-sovelluksesta löytyvälle valmiille näkymälle. Nämä sivupohjat olivat hyvin yksinkertaisia, koska ne käytännössä sisälsivät vain Djangon valmiita lomakkeita. Halusin tyylitellä lomakkeet käyttäen Bootstrap-kirjaston luokkia. Tein *generic_form.html*-sivupohjan, joka tyylitteli jokaisen lomakkeen kentän *for*-silmukkaa apuna käyttäen. Ilman tätä lomakkeiden kaikki kentät olisi pitänyt tyylitellä manuaalisesti. Alla esimerkkinä sivuston sisäänkirjautumislomakkeen HTML-koodi ja kuva valmiista lomakkeesta (Kuva 12).

```
<!-- accounts/registration/login.html -->
{% extends "base.html" %}
{% block content %}
<div class="container max-width-small">
    <h1 class="mb-5">Login</h1>
    <form action="" method="post">
        {% csrf_token %}
        {{ form.non_field_errors }}
        {% include "registration/generic_form.html" %}
        <button type="submit" class="btn btn-md btn-primary">
```

```
            Login
            </button>
        </form>
        <small>Don't have an account yet?
            <a href="{% url 'register' %}">Sign Up</a>
        </small>
        <br>
        <small>
            <a href="{% url 'password_reset' %}">
                Forgot your password?
            </a>
        </small>
</div>
{% endblock content %}
```



Kuva 12. Kirjautumiseen käytetty lomake.

7.9 Rekisteröinti

Uuden käyttäjätilin rekisteröinti piti tehdä itse, koska sitä ei ollut valmiiksi *auth*-kirjastossa. Ominaisuus tarvitsi näkymän, lomakkeen, sivupohjan ja URL-määrityksen. Hyödynsin ominaisuuden teossa Djangon valmista *UserCreationForm*-lomaketta.

7.9.1 Lomake

Aloitin tekemällä uuden lomakkeen *forms.py*-tiedostoon. Määritin lomakkeen laajentamaan *UserCreationForm*-lomaketta. Lomakkeeseen tulevat kentät piti erikseen määrittää *Meta*-luokkaan kuuluvan *fields*-muuttujan avulla. Käyttäjän sähköposti piti erikseen hakea lomakkeen lähettämästä datasta ja määrittää se *user*-objektin *email*-kentän arvoksi. Tein tämän automaattisesti kutsuttavaa *save*-funktiota käyttäen. *Username*, *password1* ja *password2* tallentuivat automaattisesti *UserCreationForm*-lomakkeen ansiosta.

```
# accounts.forms.py
class RegisterForm(UserCreationForm):
    email = forms.EmailField(required=True)

    class Meta:
        model = User
        fields = (
            "username",
            "email",
            "password1",
            "password2"
        )

        def save(self, commit=True):
            user = super(RegisterForm, self).save(commit=False)
            user.email = self.cleaned_data["email"]
            if commit:
                user.save()
            return user
```

7.9.2 Näkymä

Lomake tarvitsi sitä käyttävän näkymän. Näkymän olisi voinut tehdä käyttämällä *generic*-näkymiin kuuluvaa *FormView*-näkymää, mutta päädyin kuitenkin tekemään sen itse, koska se oli yksinkertainen. Näkymälle piti määrittää funktiot *get*- ja *post*-metodeja varten, koska täytetyn lomakkeen data lähetettäisiin näkymälle *POST*-metodia käyttäen.

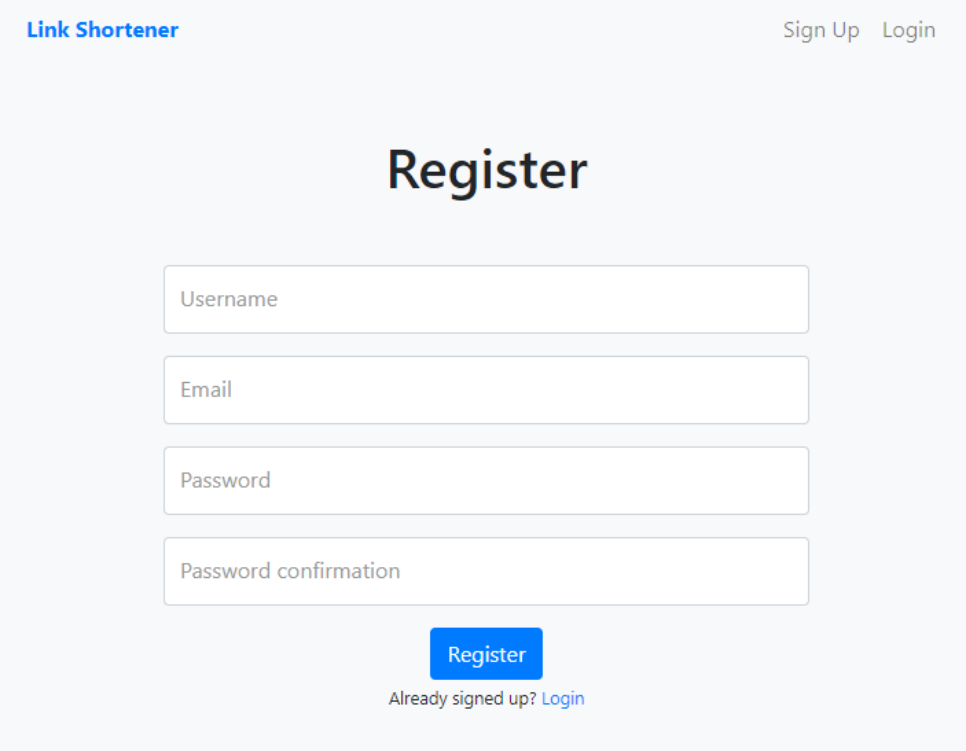
```
# accounts.views.py
class RegisterView(View):
    def get(self, request, *args, **kwargs):
        form = RegisterForm()
        context = {
            "form": form
        }
        return render(
            request,
            "registration/register.html",
            context
        )

    def post(self, request, *args, **kwargs):
        form = RegisterForm(request.POST)
        context = {
            "form": form
        }
        if form.is_valid():
            form.save()
            return redirect(reverse("login"))
        return render(
            request,
            "registration/register.html",
            context
        )
```

7.9.3 Sivupohja

Tein uuden sivupohjan *accounts/registration*-kansioon. Käytin sivupohjan teossa apuna aiemmin luomaani *generic_form.html*-sivupohjaa, joka tyylitteli lomakkeen kentät automaattisesti.

```
<!-- accounts/registration/register.html -->
{% extends "base.html" %}
{% block content %}
<div class="container max-width-small">
    <h1 class="mb-5">Register</h1>
    <form action="" method="post">
        {% csrf_token %}
        {{ form.non_field_errors }}
        {% include "registration/generic_form.html" %}
        <button type="submit" class="btn btn-md btn-primary">
            Register
        </button>
    </form>
    <small>
        Already signed up?
        <a href="{% url "login" %}">
            Login
        </a>
    </small>
</div>
{% endblock content %}
```

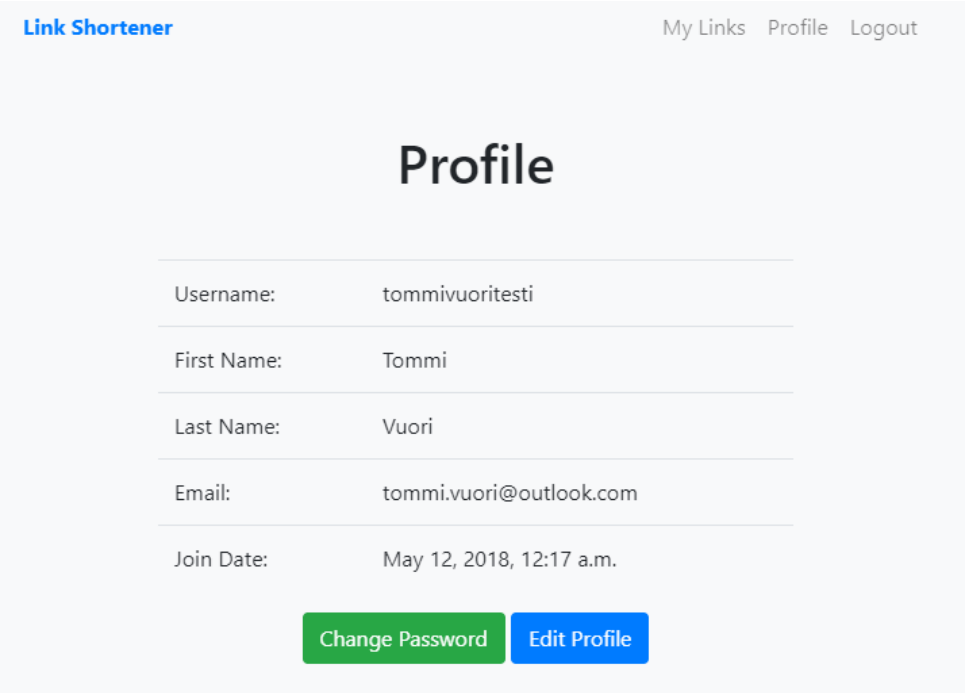


Kuva 13. Käyttäjätilin tekoon käytetty lomake.

7.10 Profiili

Tein kirjautuneita käyttäjiä varten näkymän, josta näkyi käyttäjätilien tiedot (Kuva 14). Lisäsin samaan näkymään napit käyttäjätietojen muokkausta ja salasanan vaihtoa varten. Yksinkertaisen *ProfileView*-näkymän ainut tehtävä oli palauttaa käyttäjälle sivupohja. *LoginRequiredMixin*-luokasta periytymällä näkymään pääsi ainoastaan sisäänkirjautuneet käyttäjät.

```
# accounts.views.py
class ProfileView(LoginRequiredMixin, View):
    def get(self, request, *args, **kwargs):
        return render(request, "accounts/profile.html")
```



Kuva 14. Profiilisivu.

7.11 Käyttäjätilin muokkaus

Tein lomakkeen, jonka avulla käyttäjät pystyivät muokkaamaan omia tietojaan, kuten etunimeä tai sähköpostiosoitetta. Käytin apuna Djangon valmista *UserChangeForm*-lomaketta, joka on tarkoitettu käyttäjätilin tietojen muokkaamiseen. Lomake oli muuten yksinkertainen, mutta salasanakenttä piti poistaa *super*-funktiota apuna käyttäen. Tein salasanan vaihdon erikseen käyttäen Djangon omaa *PasswordChangeForm*-lomaketta.

```
# accounts.forms.py
class EditForm(UserChangeForm):
    def __init__(self, *args, **kwargs):
        super(EditForm, self).__init__(*args, **kwargs)
        del self.fields["password"]
    class Meta:
        model = User
```

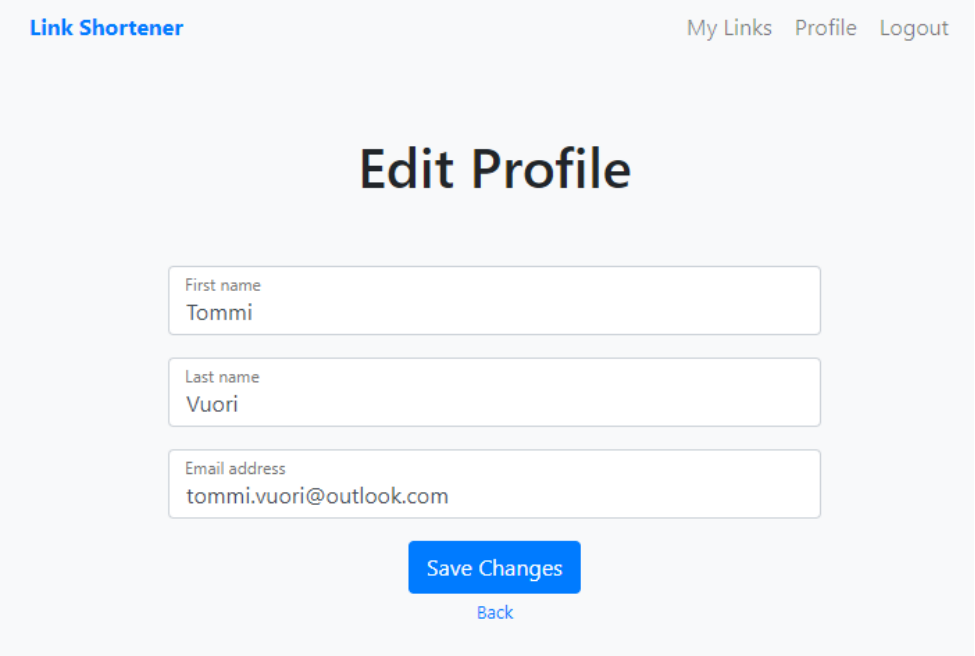
```
        fields = (
            "first_name",
            "last_name",
            "email",
            "password"
        )
```

Näkymän avulla asetin lomakkeen kenttien arvot oletuksena käyttäjän nykyisiin arvoihin määrittämällä lomakkeen *instance*-muuttujan arvoksi pyynnön lähettämän käyttäjän.

```
# accounts.views.py
class UserEditView(LoginRequiredMixin, View):
    def get(self, request, *args, **kwargs):
        form = EditForm(instance=request.user)
        context = {
            "form": form
        }
        return render(request, "accounts/user_edit.html", context)

    def post(self, request, *args, **kwargs):
        form = EditForm(request.POST, instance=request.user)
        context = {
            "form": form
        }
        if form.is_valid():
            form.save()
            return redirect(reverse("profile"))
        return render(request, "accounts/user_edit.html", context)
```

Tein sivunpohjan samanlailla, kuin aiemmatkin lomakkeet. Kenttien arvot olivat oletuksena käyttäjän nykyiset arvot (Kuva 15).



Kuva 15. Käyttäjätilin muokkaus.

7.12 Shortener-sovellus

Tein uuden sovelluksen nimellä *shortener*. Tämä sovellus tuli sisältämään kaikki linkin lyhentämiseen liittyvät asiat *ShortenedURL*-mallista hallintapaneeliin. Uusi sovellus luotiin alla olevalla komennolla.

```
python manage.py startapp shortener
```

Sovellus piti vielä aktivoida lisäämällä se settings.py-tiedostossa olevaan INSTALLED_APPS-listaan.

```
# settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'accounts',
    'shortener',
]
```

7.13 ShortenedURL-malli

Tein uuden mallin *models.py*-tiedostoon. Tarvitsin mallille vain neljä kenttää. Alkuperäisen osoitteen, lyhennetyn osoitteen ja päivämäärän lisäksi lisäsin kentän käyttäjää varten, joka linkin lyhensi. Tämän kentän avulla pystyin myöhemmin hakemaan vain kirjautuneen käyttäjän tekemät linkit.

```
# shortener.models.py
from django.db import models
from django.contrib.auth.models import User

class ShortenedUrl(models.Model):
    original_url = models.CharField(max_length=500)
    shortcode = models.CharField(max_length=50, unique=True)
    date = models.DateTimeField(auto_now_add=True)
    owner = models.ForeignKey(
        User,
        on_delete=models.CASCADE
    )
```

Lisäsin malliin vielä kaksi funktiota. *__str__*-funktio on Python-funktio, jota kutsumaan automaattisesti, kun objektia halutaan kuvata merkkijonona. *Get_absolute_url*-funktiota tulin käyttämään myöhemmin uudelleenohjauksen teossa. Asetin molemmat funktiot palauttamaan *original_url*-arvon.

```
def __str__(self):
    return self.original_url

def get_absolute_url(self):
    return self.original_url
```

7.14 Osoitteen lyhennys

Osoitteen lyhentämistä varten piti tehdä algoritmi, näkymä, lomake ja sivupohja. Suunnittelin lomakkeen sijaitsevan suoraan etusivulla, joten sen näyttämä sivupohja toimisi samalla sovelluksen etusivuna. Lomake tuli sisältämään kaksi kenttää. Ensimmäiseen kenttään syötettiin alkuperäinen osoite ja toiseen käyttäjä sai halutessaan itse määrittää lyhennetyn linkin. Jos lyhennettyä linkkiä ei erikseen määritetty, sovellus generoi satunnaisen.

7.14.1 Algoritmi

Osoitteen lyhennys -algoritmi oli erittäin yksinkertainen. Generoin satunnaisesti 8-merkkisen merkkijonon. Jos generoitu merkkijono oli jo käytössä, sovellus generoi uuden merkkijonon, kunnes se oli uniikki. Tein funktion, jonka avulla alkuperäinen osoite lyhennetään. Funktion olisi voinut laittaa näkymään tai malliin, mutta halusin pitää ne mahdollisimman siisteinä ja yksinkertaisina. Päätin tehdä utils.py-tiedoston funktiota varten. En ollut vielä varma, milloin ja missä tulisin tätä funktiota kutsumaan.

```
# utils.py
import random
import string

def create_shortcode(instance):
    new_shortcode = ""
    characters = string.ascii_letters + string.digits
    # add one random character at a time
    for x in range(8):
        new_shortcode += random.choice(characters)
    # check that it is unique
    MyClass = instance.__class__
    shortcode_exists = MyClass.objects.filter(
        shortcode=new_shortcode
    ).exists()
    # if shortcode exists, create a new one
    if shortcode_exists:
        return create_shortcode(instance)
    return new_shortcode
```

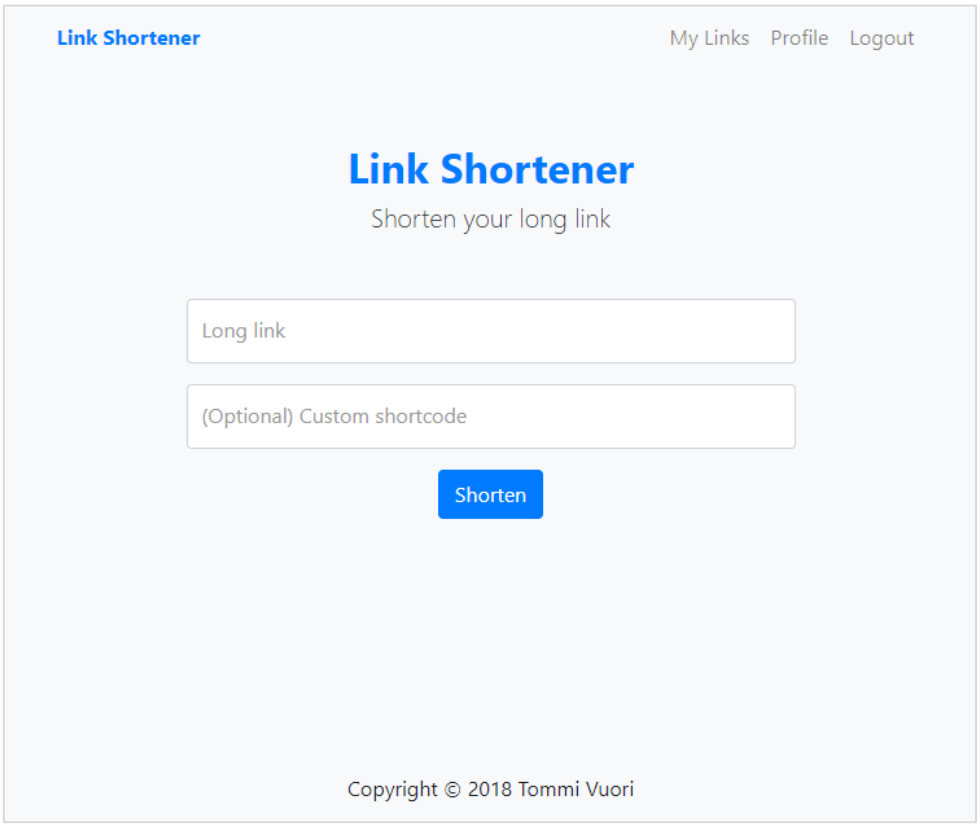
7.14.2 Näkymä

Tein lomaketta varten uuden *CreateView*-näkymästä periytyvän näkymän. *CreateView*-näkymän käyttö vähensi kirjoitettavan koodin määrää, koska lomakkeen kentät ja niiden validaattorit tulivat suoraan aiemmin luodusta mallista. Annoin lomaketta käyttävän näkymän nimeksi *HomeView*, koska se tuli toimimaan palvelun etusivuna. Kaikkien kenttien sijaan valitsin lomakkeeseen ainoastaan alkuperäisen pitkän osoitteen ja lyhyelle osoitteelle tulevan päätteen. Kun osoitteen lyhennys oli valmis ja tallennettu tietokantaan, halusin uudelleenohjata käyttäjän detail-näkymään, josta pystyi esimerkiksi kopioimaan lyhennetyn osoitteen.

```
# shortener.forms.py
class HomeView(CreateView):
    model = ShortenedUrl
    success_url = ""
    template_name_suffix = "_create"
    fields = [
        "original_url",
        "shortcode"
    ]
```

Luotua objektia piti vielä muuttaa ennen sen tallentamista tietokantaan. Muun muassa sen tekijä piti määrittää *owner*-kenttään. Lisäksi uudelleenohjausta varten tarvittava URL-osoite piti määrittää luodun objektin primääriavainkentän arvon avulla. Tein myös lyhyen osoitteen generoinnin tässä vaiheessa, jos käyttäjä ei ollut sitä manuaalisesti määrittänyt. Käytin generointiin aiemmin luomaani *utils.py*-tiedostossa olevaa *create_shortcode*-funktiota. Tein nämä muutokset automaattisesti ajettavan *form_valid*-funktion avulla, jota kutsuttiin, kun lomake läpäisi sille asetetun validoinnin. Onnistuneen lyhentämisen jälkeen näkymä uudelleenohjasi käyttäjän *Detail*-näkymään *redirect*-funktion avulla.

```
# shortener.views.py
def form_valid(self, form):
    obj = form.save(commit=False)
    if (self.request.user.is_authenticated == False
        and obj.shortcode != None):
            raise PermissionDenied
    if not obj.shortcode:
        obj.shortcode = create_shortcode(obj)
    if self.request.user.is_authenticated:
        obj.owner = self.request.user
    obj.save()
    self.success_url = reverse_lazy(
        "url-detail",
        kwargs={
            "pk":obj.pk
        }
    )
    return redirect(self.success_url)
```



Kuva 16. Sovelluksen etusivu, jossa oli lyhennykseen käytettävä lomake.

7.15 Detail-näkymä

Detail-näkymä oli näkymä, johon käyttäjä ohjattiin linkin lyhentämisen jälkeen. Näkymä näytti alkuperäisen pitkän osoitteen, lyhennetyn osoitteen ja napin, jonka avulla lyhennetyn osoitteen pystyi automaattisesti kopioimaan. Käytin näkymässä Djangosta valmiiksi löytyvää *DetailView*-näkymää laajentamalla sitä. Näkymästä tuli hyvin yksinkertainen, koska laajennettu *DetailView*-näkymä toimi lähes sellaisenaan. Oikean objektin *DetailView*-näkymä haki URL-määrityksen mukana tulevan primääriavaimen perusteella.

```
# shortener.views.py
class ShortenedUrlDetailView(DetailView):
  model = ShortenedUrl
```

Detail-näkymän URL-määritys ja sivupohja:

```
# shortener.urls.py
path(
    "shortened-url/<int:pk>/",
    ShortenedUrlDetailView.as_view(),
    name="shortened_url_detail"
)

<!-- shortener/shortenedurl_detail.html -->
{% extends "base.html" %}
{% block content %}
```

```
<h1>{{ object.shortcode }}</h1>
<h2>Original URL:</h2>
<p>{{ object.original_url }}</p>
{% endblock content %}
```

7.16 Lyhyiden osoitteiden uudelleenohjaus

Uudelleenohjausta varten piti tehdä URL-määritys ja näkymä. Aloitin näkymästä, koska sitä käytettiin URL-määrityksen *path*-funktiossa. Oikea alkuperäinen URL haettiin URL-määrityksessä olleen dynaamisen osan avulla. Näkymässä tähän arvoon pääsi käsiksi *kwargs*-listan avulla, joka sisälsi *request*-objektin mukana tulleet avainsana-argumentit. Jos *shortcode*-muuttujaa vastaavaa objektia ei löytynyt, näkymä palautti 404-virheen, joka tarkoitti, ettei haluttua resurssia löytynyt. Lopuksi näkymä uudelleenohjasi käyttäjän *redirect*-funktion avulla.

```
# shortener.views.py
from django.shortcuts import get_object_or_404, redirect
from django.views import View
from .models import ShortenedUrl

class RedirectShortenedUrl(View):
    def get(self, request, *args, **kwargs):
        destination_url = get_object_or_404(
            ShortenedUrl,
            shortcode=kwargs["shortcode"]
        )
        return redirect(destination_url)
```

Tein sovelluskohtaiseen *urls.py*-tiedostoon uuden URL-määrityksen. Määritys kaappaa minkä tahansa merkkijonon, joka tulee suoraan sivuston domainin jälkeen, ja ohjaa *RedirectShortenedUrl*-näkymään.

```
# shortener.urls.py
from django.urls import path
from .views import RedirectShortenedUrl

urlpatterns = [
    path("<str:shortcode>/", RedirectShortenedUrl.as_view()),
]
```

7.17 Lyhennettyjen osoitteiden hallintapaneeli

Tein sovellukseen näkymän, joka listasi käyttäjän lyhentämät osoitteet. Samalla näkymä teki napit osoitteiden kopioimista ja poistamista varten. Annoin näkymälle nimeksi *MyUrlsView*. Näkymä haki kaikki *ShortenedUrl*-mallin objektit, joiden tekijä oli *request.user*, eli pyynnön lähettänyt käyttäjä. Lisäsin nämä objektit *context*-dataan, joka lähetettiin sivupohjalle.

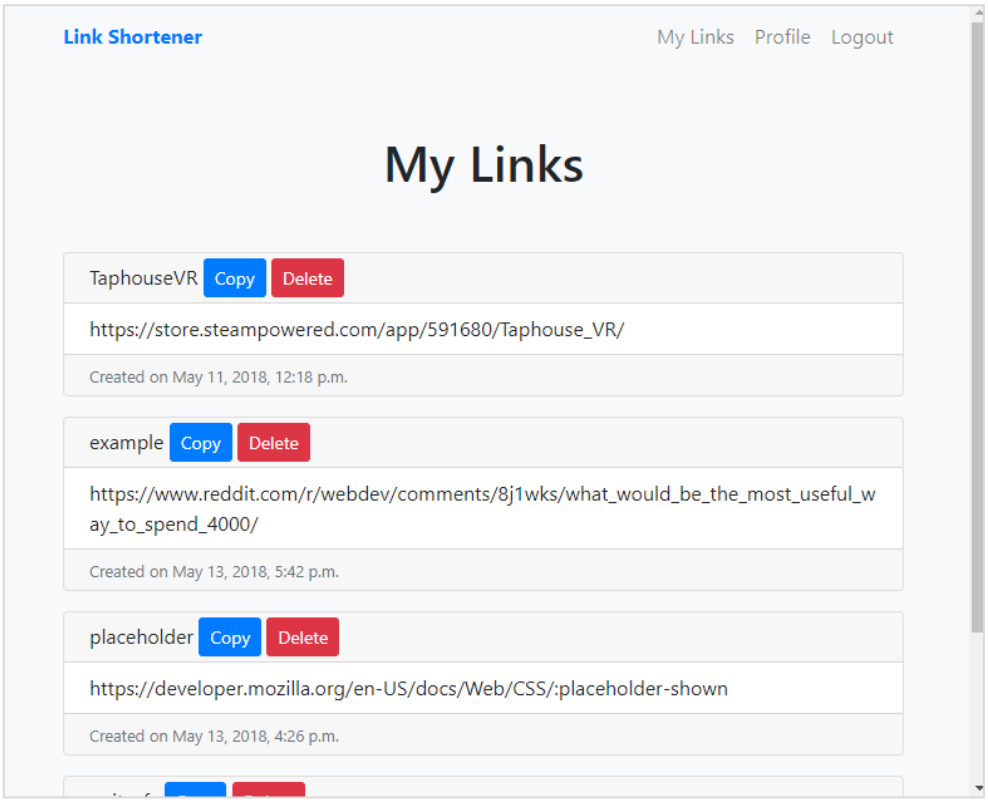
```
# shortener.views.py
class MyUrlsView(LoginRequiredMixin, View):
    def get(self, request):
        my_urls = ShortenedUrl.objects.all().filter(
            owner=request.user
```

```
        )
        context = {
            "my_urls": my_urls
        }
        return render(request, "shortener/my_urls.html", context)
```

Käytin sivupohjassa for-silmukkaa jokaisen lyhennetyn linkin listaamiseen. Järjestin osoitteet *shortcoden* perusteella aakkosjärjestykseen.

```
<!-- shortener/my_urls.html-->
{% if my_urls %}
    {% for link in my_urls|dictsort:"shortcode" %}
        <!-- Shortened links -->
    {% endfor %}
{% else %}
    <p>You have not shortened any links yet.</p>
{% endif %}
```

Koska järjestäminen toimi ASCII-koodien perusteella, isolla kirjaimella alkavat linkit menivät listan ensimmäisiksi (Kuva 17). Se ei ollut aivan sitä, mitä halusin, mutten nähnyt tarpeelliseksi korjata asiaa.



Kuva 17. Linkkien hallintaan tarkoitettu sivu, johon oli listattu käyttäjän lyhentämät osoitteet.

7.18 Lyhennettyjen osoitteiden poistaminen

Tein uuden näkymän osoitteiden poistamista varten. Käytin näkymän pohjalla *DeleteView*-näkymää, joka oli tarkoitettu malleihin liittyvien objektien poistamiseen. Lisäsin *delete*-funktioon tarkistuksen, joka varmisti, että

poistettava linkki oli sillä hetkellä kirjautuneen käyttäjän tekemä. Poiston onnistumisen jälkeen näkymä uudelleenohjasi käyttäjän takaisin hallintapaneeliin. *LoginRequiredMixin*-luokkaa apuna käyttäen kirjautumattomat käyttäjät eivät päässeet näkymään.

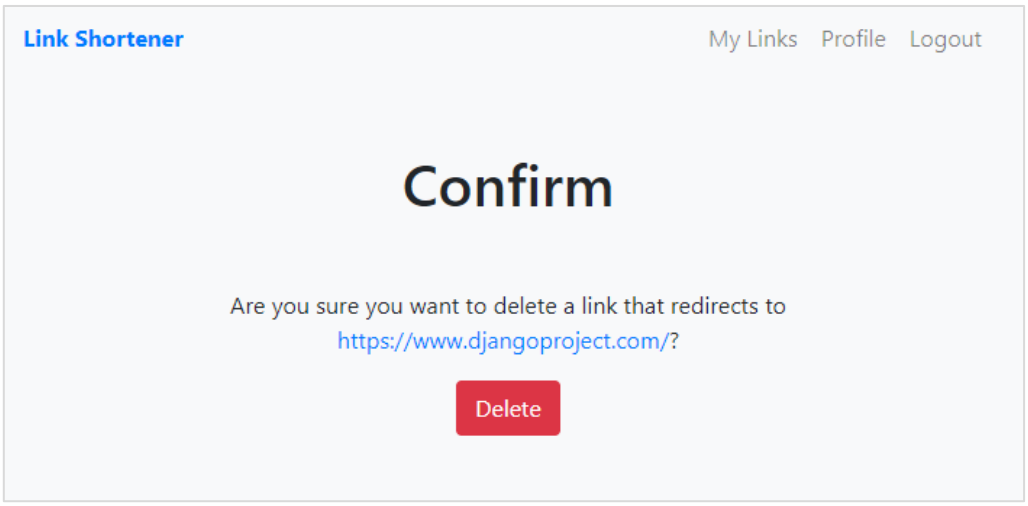
```
# shortener.views.py
class ShortenedUrlDeleteView(LoginRequiredMixin, DeleteView):
    model = ShortenedUrl
    success_url = reverse_lazy("my-urls")

    def delete(self, request, *args, **kwargs):
        self.object = self.get_object()
        if self.object.owner == request.user:
            self.object.delete()
            return HttpResponseRedirect(self.get_success_url())
        else:
            raise Http404
```

Tein näkymää varten uuden URL-määrityksen. Määritys sisälsi dynaamisen osan, jonka perusteella *ShortenedUrlDeleteView*-näkymä valitsi oikean objektin. Käytin objektin primääriavainta valintaperusteena.

```
# shortener.urls.py
path(
    "shortened-url/<int:pk>/delete/",
     ShortenedUrlDeleteView.as_view(),
     name="url-delete"
)
```

DeleteView-näkymä tarvitsi sivupohjan, joka sisälsi POST-metodia käyttävän lomakkeen, jolla poistaminen vahvistettiin. Sivupohjan nimen piti olla muotoa *modelname_confirm_delete.html*. Valmis lomake sisälsi ainoastaan *submit*-tyyppiä olevan napin (Kuva 18).



Kuva 18. Lomake, jolla vahvistettiin lyhennetyn osoitteen poistaminen.

7.19 Linkin kopioiminen

Lisäsin vielä linkin kopioimiseen tarkoitettuihin nappeihin JavaScript-koodin, jonka avulla linkin kopioiminen leikepöydälle tapahtui. Käytin koodissa jQuery-kirjastoon kuuluvia toimintoja, koska se oli jo projektissa Bootstrap-kirjaston takia. Tein uuden JavaScript-tiedoston projektin juuressa olevan *static*-kansion sisällä olevan *js*-kansion sisälle. Annoin tiedostolle nimeksi *main.js*. Lisäsin tämän samalla *base.html*-sivupohjan *body*-elementin loppuun. Koodi loi väliaikaisen elementin, jonka sisällöksi se määritti lyhennetyn linkin. Elementin sisällön kopioimisen jälkeen elementti poistettiin *remove*-funktiolla.

```
// main.js
window.onload = function () {
    // Get all the elements that match the selector as arrays
    var shortenedUrlCopyButtons = Array.prototype.slice.call(
        document.querySelectorAll(".shortened-url-copy-button")
    );
    var shortenedUrls = Array.prototype.slice.call(
        document.querySelectorAll(".shortened-url")
    );

    // Loop through the button array
    shortenedUrlCopyButtons.forEach(function(btn, idx){
      btn.addEventListener("click", function(){
        var $temp = $("<input>");
        $("body").append($temp);
        $temp.val(document.location.host + "/" + shortene-
dUrls[idx].innerHTML).select();
        document.execCommand("copy");
        $temp.remove();
      });
    });
}
```

8 YHTEENVETO

Projektin tavoitteena oli toteuttaa toimiva web-sovellus Djangolla, jolla pystyisi lyhentämään pitkiä linkkejä. Linkkien lyhentämisen lisäksi sovelluksen vaatimuksina oli tuki käyttäjätilien tekoon ja niiden hallinnointiin. Opinnäytetyön tuloksena syntynyt sovellus vastasi sille asetettuja tavoitteita ja vaatimuksia. Yhtenä esimerkkiprojektin tarkoituksena oli perehtyä siihen, miten hyvin Django toimii vielä nykyisin sovellusten teossa, kun asiakaspuolen sovelluskehykset ovat alkaneet nousta suosiossa. Yhä useammat suositut palvelut ovat siirtyneet käyttämään JavaScript-sovelluskehyksiä sulavamman ja reaktiivisemman käyttökokemuksen takaamiseksi. Mielestäni Django kuitenkin soveltui tässä työssä tehtyyn sovellukseen erinomaisesti.

8.1 Haasteet

Minulla oli jonkin verran kokemusta Djangosta jo ennen opinnäytetyön aloittamista, joten osasin melko hyvin hyödyntää Djangon valmiiksi tarjoamia ominaisuuksia, kuten esimerkiksi Djangon valmiita näkymiä apuna omien näkymien teossa. Ilman näitä lähtötietoja ylipäätään näiden ominaisuuksien löytäminen olisi tuottanut haasteita tai olisin tehnyt jokaisen näkymän täysin tyhjästä.

Törmäsin myös ongelmiin työtä tehdessä. Yksi työssä vastaan tullut ongelma liittyi lomakkeiden tyylittelyyn. Mietin pitkään, miten pystyisin järkevästi tyylittelemään lomakkeita ilman, että niistä jokainen pitäisi kirjoittaa manuaalisesti kenttä kerrallaan. Tähänkin kuitenkin löytyi vastaus lähes suoraan Djangon omasta dokumentaatiosta, kunhan vain osasi hakea oikealla hakusanalla. Toinen ongelma liittyi lomakkeeseen, jolla päivitettiin käyttäjätietoja. Lomake piti sisällään oletuksena kaikki mahdolliset rivit käyttäjänimestä liittymispäivään ja niitä pystyi vapaasti muokkaamaan. Turhat kentät sai melko helposti poistettua mutta salasanakentän poistamisen jälkeen lomake ei toiminut ollenkaan. Pienen tutkimisen jälkeen se piti poistaa vasta lomakkeen alustuksen jälkeen.

8.2 Jatkokehitys

Sovellus on ominaisuuksiltaan tällä hetkellä yksinkertainen ja sitä olisi helppo jatkaa joka osa-alueella. Ennen työn aloitusta suunnittelin, että lopullinen versio sisältäisi jokaiselle lyhennetylle linkille oman sivun, joka sisältäisi muun muassa klikkausten määrän, ajankohdan ja muuta tietoa. Tämän datan olisi voinut myös visualisoida käyttäen esimerkiksi Chart.js-kirjastoa. Tämä ominaisuus jäi kuitenkin puuttumaan, koska sovellus olisi mielestäni kasvanut liian isoksi opinnäytetyötä varten, eikä kyseinen ominaisuus kuitenkaan ole sovelluksen toiminnan kannalta elintärkeä.

Toinen idea olisi korvata Djangon oma sivupohjajärjestelmä jollakin JavaScript-sovelluskehyksellä, kuten Reactilla. Näin esimerkiksi linkkien hallintapaneelista tulisi hieman sujuvampi käyttää. Reactin tai muun JavaScript-sovelluskehyksen käyttö lisäisi lopulta kuitenkin varsin vähän lisäarvoa sovelluksen toimintaan. Olisi eri asia, jos sovellus sisältäisi paljon koko ajan muuttuvia dynaamisia elementtejä.

Sovelluksen ulkoasu on tehty käyttäen enimmäkseen Bootstrap-kirjaston oletustyylejä. Ne toimivat, mutta on mahdollista, että jokin toinen sovellus näyttäisi ulkoisesti lähes identtiseltä. Sovellusta ei ole myöskään optimoitu isoille resoluutioille, joita esimerkiksi linkkien hallintapaneeli ei osaa järkevästi hyödyntää.

LÄHTEET

Angular (n.d.). AngularJS to Angular Concepts: Quick Reference. Viitattu 28.4.2018.
https://angular.io/guide/ajs-quick-reference

Akshar (2015). Understanding Django Middlewares. Viitattu 12.5.2018.
https://www.agiliq.com/blog/2015/07/understanding-django-middlewares/

Atlassian (n.d.). What is Git. Viitattu 28.4.2018.
https://www.atlassian.com/git/tutorials/what-is-git

Basu, S. (2013). Ruby of Rails Study Guide: The History of Rails. Viitattu 14.4.2018.
https://code.tutsplus.com/articles/ruby-on-rails-study-guide-the-history-of-rails--net-29439

Brown, K. (2016). What is GitHub, and What Is It Used For? Viitattu 28.4.2018.
https://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/

Django Documentation (n.d.). Making Queries. Viitattu 2.5.2018.
https://docs.djangoproject.com/en/2.0/topics/db/queries/

Django Documentation (n.d.). The Django template language. Viitattu 2.5.2018.
https://docs.djangoproject.com/en/2.0/ref/templates/language/

Django Documentation (n.d.). Why does this project exist? Viitattu 7.5.2018.
https://docs.djangoproject.com/en/dev/faq/general/#why-does-this-project-exist

Freitas, V. (2017). Class-Based Views vs. Function-Based Views. Viitattu 5.5.2018.
https://simpleisbetterthancomplex.com/article/2017/03/21/class-based-views-vs-function-based-views.html

Gavigan, D. (2018). The History of Angular. Viitattu 5.5.2018.
https://medium.com/the-startup-lab-blog/the-history-of-angular-3e36f7e828c7

GitHub (2017). The fifteen most popular languages on GitHub. Viitattu 15.5.2018.
https://octoverse.github.com/

Google for Education (2017). Python Introduction. Viitattu 15.4.2018.
https://developers.google.com/edu/python/introduction

Google for Education (2017). Code Checked at Runtime. Viitattu 28.3.2018.
https://developers.google.com/edu/python/introduction#Code%Checked%at%Runtime

HotFrameworks (2018). Web framework rankings. Viitattu 6.4.2018.
https://hotframeworks.com/#top-frameworks

IEEE Spectrum (2017). The 2017 Top Programming Languages. Viitattu 5.5.2018.
https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

Ndegwa, A. (2016). What is a Web Application? Viitattu 14.4.2018.
https://www.maxcdn.com/one/visual-glossary/web-application/

Niemi, T. (2015). Sovelluskehys – Verkkoräätälin työkalupakki. Viitattu 14.4.2018.
https://www.sofokus.com/fi/blogi/sovelluskehys-verkkoraatalin-tyokalupakki/

Microsoft (2018). Introduction to ASP.NET Core. Viitattu 28.3.2018.
https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.1

O'Brien, J. (2016). The History of Laravel. Viitattu 14.4.2018.
https://medium.com/vehikl-news/a-brief-history-of-laravel-5d55970885bc

Python.org (2013). Python Introduction. Viitattu 2.5.2018.
https://www.python.org/dev/peps/pep-0008/#introduction

Python Wiki (n.d.). For Loops. Viitattu 5.5.2018.
https://wiki.python.org/moin/ForLoop

Python Wiki (n.d.). Why is Python a dynamic language and also a strongly typed language. Viitattu 30.3.2018.
https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language

Reactjs (n.d.). Introducing JSX. Viitattu 5.5.2018.
https://reactjs.org/docs/introducing-jsx.html

Ronacher, A. (2017). What does "micro" mean? Viitattu 14.3.2018.
http://flask.pocoo.org/docs/0.12/foreword/#what-does-micro-mean

TheDjangoBook (n.d.). Model-View-Controller Design Pattern. Viitattu 15.4.2018.
http://djangobook.com/model-view-controller-design-pattern/

TheDjangoBook (n.d.). Defining Django Models in Python. Viitattu 7.5.2018.
https://djangobook.com/django-models/#defining-django-models-in-python

TIOBE (2018). TIOBE Index for April 2018. Viitattu 15.4.2018.
http://www.tiobe.com/tiobe-index/

ValueCoders (2018). Why Laravel Is The Best PHP Framework In 2018? Viitattu 14.4.2018.
https://www.valuecoders.com/blog/technology-and-apps/laravel-best-php-framework-2017/

Visual Studio Code Documentation (n.d.). Why did we build Visual Studio Code? Viitattu 28.4.2018.
https://code.visualstudio.com/docs/editor/whyvscode

Vega, J. (2017). Client-side vs. server-side rendering: why it's not all black and white. Viitattu 14.4.2018.
https://medium.freecodecamp.org/what-exactly-is-client-side-rendering-and-hows-it-different-from-server-side-rendering-bd5c786b340d

Vuejs (n.d.). What is Vue.js? Viitattu 8.5.2018.
https://vuejs.org/v2/guide/

Vuejs (n.d.). Comparison with Other Frameworks. Viitattu 8.5.2018.
https://vuejs.org/v2/guide/comparison.html

Wappalyzer (n.d.). Websites using Express. Viitattu 8.5.2018.
https://www.wappalyzer.com/technologies/express