jamk

Headless-toteutuksen kehitys WordPress- ja Next.js-ympäristössä

Petteri Mattila

Opinnäytetyö, AMK
Toukokuu 2022
Tietojenkäsittely ja tietoliikenne
Tradenomi (AMK), tietojenkäsittelyn tutkinto-ohjelma

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Mattila Petteri

Headless-toteutuksen kehitys WordPress- ja Next.js-ympäristössä

Jyväskylä: Jyväskylän ammattikorkeakoulu. Toukokuu 2022, 62 sivua.

Tietojenkäsittely ja tietoliikenne. Tietojenkäsittelyn tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Opinnäytetyön tutkimuksen tarkoituksena oli selvittää headless-toteutuksen positiiviset ja negatiiviset vaikutukset sekä selvittää, mitä asioita toteutuksessa pitäisi huomioida, jotta kehitystyö voitaisiin toteuttaa onnistuneesti. Tarvittaessa kehitystyössä tehdystä toteutuksesta voidaan kehittää sovelluspohja, jota voidaan hyödyntää muissa headless-toteutuksissa. Tutkimuksessa ei tutkittu pelkästään opinnäytetyössä käytettyä ympäristöä, vaan tutkimus tehtiin laajemmalla näkökulmalla, jonka myötä tuloksia voidaan soveltaa muissa toteutuksissa, jotka eivät käytä samoja teknologioita kuin opinnäytetyö. Opinnäytetyön toimeksiantajana toimi Frostigon Oy.

Headless-toteutuksessa tehtiin käyttöliittymä käyttäen Next.js-frameworkiä, joka pystyy hakemaan tietoja WordPress-sisällönhallintajärjestelmästä. WordPress-ympäristöön lisättiin tarvittavat lisäosat: WPGraphQL, Custom Post Type UI ja Advanced Custom Fields. Kyseiset lisäosat mahdollistivat tiedon välityksen sisällönhallintajärjestelmän ulkopuolelle halutussa muodossa. Lopullisena toteutuksena opinnäytetyössä tehtiin sovelluspohja, jolla voidaan luoda suhteellisen lyhyessä ajassa toimiva demosovellus. Sovellusta voidaan jatkokehittää muokkaamalla templaatteja, tyylittelyjä ja lisäämällä sovelluksen vaatimat ominaisuudet. Kyseisillä muutoksilla voidaan saavuttaa lopullinen headless-toteutus.

Tutkimus toteutettiin laadullisena tutkimuksena, jonka tuloksina saatiin selville toteutuksen mahdolliset positiiviset ja negatiiviset vaikutukset, joiden avulla voidaan perustella, onko headless-toteutustapa hyödyllinen tietyssä tilanteessa. Headless-toteutuksen luonteen vuoksi tutkimuksessa ei pystytty vastaamaan minkälaisissa tilanteissa toteutusta ei voitaisi käyttää tai missä sitä pitäisi käyttää. Tutkimuksessa tarkasteltiin eri teknologioiden hyviä ja huonoja puolia perinteisessä- sekä headless-toteutuksessa. Kyseistä tietoa voidaan hyödyntää sovelluksen toteutuksen suunnittelussa.

Avainsanat (asiasanat)

WordPress, Next.js, headless, web-kehitys, sovelluskehitys

Muut tiedot (salassa pidettävät liitteet)

Ei salassa pidettäviä liitteitä.

jamk

Description

Mattila Petteri

Development of headless implementation in WordPress and Next.js environment

Jyväskylä: JAMK University of Applied Sciences, May 2022, 62 pages.

Information and communication technologies. Business Information Technology. Bachelor's thesis.

Permission for open access publication: yes

Language of publication: Finnish

Abstract

The purpose of the thesis was to find out the positive and negative effects of headless implementation and to find out what issues should be taken into account in order to successfully create headless environment. If necessary, the developed application can be developed further into an application base that can be used in other headless implementations. The study did not only examine the environment used in the thesis, but also took a broader perspective, allowing the results to be used in other headless implementations that do not use the same technologies as the thesis. The thesis was commissioned by Frostigon Oy.

The headless implementation contains a user interface which was created using Next.js framework, which can retrieve information from WordPress. WPGraphQL, Custom Post Type UI and Advanced Custom Fields plugins were added to the WordPress environment, which allowed applications outside of the content management system to retrieve needed content in the desired format. The final work of the thesis was an application base, that can be used to create a working demo application in a relatively short time. The application can be further developed by editing templates, stylesheets and adding features required by the application. With these changes, the final headless implementation can be achieved.

The study was carried out as a qualitative study to identify the potential positive and negative effects of the implementation. The study can be used to justify whether headless implementation is beneficial in a given situation. Due to the nature of headless implementation, the study could not answer in which situations the implementation could not be used or where it should be used. The study looked at the pros and cons of different technologies in traditional and headless implementation. This information can be used to design the implementation of the application.

Keywords/tags (subjects)

WordPress, Next.js, headless, web development, software development

Miscellaneous (Confidential information)

No confidential information.

Sisältö

Kuviot

Määritelmät ja sanasto

API	Rajapinta, jonka välityksellä sovellukset voivat välittää tietoja (eng. Application Programming Interface).
Backend	Englanninkielinen termi, jolla viitataan järjestelmään, joka toimii sovelluksen taustalla. Backend-järjestelmään voi kuulua itse kirjoitetut backend-sovellukset, tietokannat ja palvelimet.
Custom Post Type	Englanninkielinen termi (lyh. CPT), joka voidaan kääntää mukautetuksi artikkelityypiksi, johon viitataan tekstissä lyhenteellä CPT. CPT on osa taksonomian tapoja luoda alaluokkia artikkeleille, joiden avulla voidaan kategorioida artikkeleita.
Framework	Englanninkielinen termi (suom. viitekehys), jolla tarkoitetaan tiedostorakennetta, jonka varaan voidaan rakentaa sovelluksia. Viitekehyksiä on kehitetty monella eri teknologialla, jonka ansiosta viitekehykset soveltuvat moneen eri sovelluskehitysprojektiin.
Frontend	Englanninkielinen termi, jolla viitataan päätelaitteella toimivaan sovellus ratkaisuun.
GraphQL	Facebookin kehittämä modernikyselykieli, jonka avulla voidaan tuoda sisältöä sovellukseen.
Headless	Toteutustapa, jossa erotetaan frontend backendistä. Toteutustavalla saavutetaan ympäristö, jonka frontend- ja backend-toteutuksia voidaan räätälöidä tarpeen mukaisesti.
JSX	React-kirjaston käyttämä JavaScript-syntaksilaajennus.

Lisäosat	Tapa tuoda WordPress-toteutukseen ominaisuuksia ilman, että ominaisuuksia pitää sisällyttää teemaan.
Mikropalvelu	Sovelluksen erillinen palvelu, johon kuuluu mm. REST API.
Next.js	Vercel-yhtiön kehittämä React-kirjastosta pohjautuva framework.
Päätepiste	Sovellukseen määritetty piste, josta siihen yhdistetyt sovellukset voivat hakea sisältöä (eng. endpoint).
REST API	Palvelu, joka tuo sisältöä sovellukseen HTTP-pyyntöjen avulla.
React	Facebookin kehittelemä JavaScript-kirjasto.
Shopify	Verkkokauppaohjelma, jonka avulla voidaan rakentaa ja perustaa verkkokauppoja. Shopify on myös kykeneväinen hallinnoimaan kaupan tuotteita ja tehtyjä tilauksia.
SQL	Kyselykieli, jolla voidaan tehdä hakuja, muutoksia ja lisäyksiä tietokantaan (eng. Structured Query Language).
Taksonomi	Tapa luokitella WordPress-artikkeleita eri kategorioihin. Kategorian voi luokitella CPT:n tai tagin avulla, jotka ovat taksonomian tapoja luoda kategorioita (eng. taxonomy).
Teemat	WordPress-toteutuksen frontend-ratkaisu, joka toteutetaan käyttäen PHP-, CSS- ja JavaScript-ohjelmointikieliä.
WordPress	Suosittu sisällönhallintajärjestelmä, joka mahdollistaa sovelluksien sisällön muokkaamisen ilman lähdekoodin muokkaamista.

1 Johdanto

Sovelluksien ylläpitämisessä hyödynnetään monenlaisia teknologioita, jotka edesauttavat niin käyttäjän, kuin ylläpitäjänkin käyttökokemusta. Näihin teknologioihin kuuluvat muun muassa perinteiset sisällönhallintajärjestelmät, jotka ovat olleet kauan osana sovelluskehityksen kokonaisuutta. Ajan myötä on todettu, etteivät perinteiset sisällönhallintajärjestelmät sovellu jokaiseen toteutukseen. Perinteiset sisällönhallintajärjestelmät eivät tue tietojen välitystä useaan päätepisteeseen ja joissakin tapauksissa perinteisiä sisällönhallintajärjestelmiä ei voida käyttää niiden luontaisen arkkitehtuurirakenteen vuoksi, mikä voi vaikuttaa miten lopullinen ratkaisu toimii käytännössä. Tämä on johtanut moderneista teknologioista pohjautuvien sisällönhallintajärjestelmien kehitykseen, joita usein kutsutaan headless-sisällönhallintajärjestelmiksi. Headless-sisällönhallintajärjestelmien kevyen luonteen vuoksi järjestelmä tulee tarvitsemaan irrallisen käyttöliittymän, johon sisällönhallintajärjestelmä tullaan yhdistämään. Moderneja headless-teknologioita käyttävän sovelluksen lopullista ympäristöä voidaan kutsua headless-ympäristöksi.

Opinnäytetyössä käsitellään headless-ympäristön toteuttamista WordPress-ympäristössä käyttäen Next.js-frameworkiä käyttöliittymän toteuttamisessa. Teknologiat on valittu suosion ja tuttavuuden perusteella; WordPress on laajasti käytetty sisällönhallintajärjestelmä, joka on monelle tuttu työelämän kautta ja Next.js on yksi suosituimmista React-kirjastosta pohjautuvista sovelluskehityksen työkaluista. Aiheen valintaan vaikuttivat aiemmista tutkimuksista kulunut aika, tietämättömyys headless-toteutuksista ja toimeksiantajan tarpeet.

WordPressissä toteutettava headless-ympäristö on suhteellisen uusi ja täten myös mielenkiintoinen aihe. Aihe herätti useamman kysymyksen sen hyödyistä ja haitoista verrattuna perinteiseen WordPress-toteutukseen. Tämän lisäksi toimeksiantajalla oli tarve siirtää perinteisellä WordPress-toteutuksella tehty sovellus headless-toteutukseen, mitä oli tarkoitus alustaa luomalla moderneilla sovelluskehitystekniikoilla toteutettu headless-ympäristön pohja.

2 Tutkimusasetelma

Luvussa alustetaan opinnäytetyön vaiheita, joihin kuuluu tavoitteiden, rajauksen, tutkimus menetelmien ja tutkimus kysymyksien esittely. Luvun jälkeen lukijalle pitäisi olla selvää miksi opinnäytetyötä tehdään, mikä on sen näkökulma ja mitä aiheita opinnäytetyö tulee käsittelemään.

2.1 Toimeksiantaja

Opinnäytetyön toimeksiantajana toimii mainostoimisto Frostigon Oy, joka on erikoistunut myös viestintätoimiston ja digitoimiston tehtäviin. Toimeksiantajan tarjontaan kuuluu sisällöntuotanto, hakukoneoptimointi, SEO-auditointi, konsultaatio, WordPress-sivujen ja lisäosien räätälöinti. Toimeksiantaja on myös asettanut tehtäväkseen edistää IT-alaa tarjoamalla ajanmukaisia vinkkejä mm. sovelluskehitykseen blogi muodossa. (Frostigon 2022.)

Toimeksiantajalla on useita asiakkaita, joiden nettisivuratkaisuna toimii perinteinen WordPress-toteutus. Suurin osa näistä toteutuksista toimivat hyvin, mutta toimeksiantajalla on yksi WordPress-toteutus, joka on muita vaativampi ja tulee tarvitsemaan useita kolmannen osapuolen tarjoamia sovellusratkaisuja, joiden lisääminen olemassa olevaan WordPress-toteutukseen ei ole ihanteellinen ratkaisu pitkällä tähtäimellä. WordPress-toteutuksen lähdekoodi tulisi todennäköisesti hyvin vaikeaksi ylläpitää eikä WordPress alustana tarjoa tarvittavaa joustavuutta sovelluskehityksessä, jonka vuoksi kyseisen WordPress-toteutuksen osalta toimeksiantaja ja asiakas ovat päätyneet headless-toteutukseen. Mahdollinen headless-toteutus tulisi lisäämään toivottua tietoturvaa, hakukone optimoinnista tulisi suoraviivaisempaa, puhtaampi lähdekoodi, parempi skaalautuvaisuus ja erikielisille käyttäjille pystyttäisiin tarjoamaan paremmin lokalisoitua sisältöä. (Smal 2022.)

2.2 Tavoitteet ja rajaus

Opinnäytetyön tutkimuksen tavoitteena on selvittää headless-toteutuksen negatiiviset ja positiiviset puolet. Tämän lisäksi tutkimuksessa selvitetään minkälaisia asioita pitäisi huomioida ennen kehitystyön aloittamista ja miten kyseiset asiat vaikuttavat kehitystyön toteuttamiseen. Tällä luodaan tukeva tietoperusta, joka tukee toteuttamisympäristön rakenteen ymmärtämistä.

Tutkimus on rajattu headless-toteutuksen oleellisiin frontend-teknologioiden käsittelyyn, joihin kuuluu Next.js-framework ja WordPress-sisällönhallintajärjestelmä. Jotta tutkimuksen sisältö olisi headless-toteutuksen osalta kattava, tutkimuksessa selvitetään muutaman esimerkin avulla, minkälaisilla teknologioilla toteutus voidaan suorittaa ja miten eri toteutustavat voivat vaikuttaa lopputulokseen.

Empiirisessä osuudessa tullaan luomaan headless-ympäristö, joka sisältää WordPress-sisällönhallintajärjestelmän ja Next.js-sovelluksen. Next.js-sovellus tulee sisältämään pelkistetyssä muodossa samoja ominaisuuksia kuin WordPressin sisältämä frontend-ratkaisu. Tämän avulla päädytään frontend-ratkaisuun, jolla voidaan korvata aiemmin WordPressissä käytössä ollut ratkaisu ilman oleellisien ominaisuuksien menettämistä. Pelkistettyä frontend-sovellusta voidaan hyödyntää muissa headless-toteutuksissa käyttämällä sovellusta sovelluspohjana, johon voidaan lisätä uuden toteutuksen vaatimia ominaisuuksia. Teoreettisen ja empiirisen osuuden jälkeen lukijalle pitäisi olla selvää minkälaisissa tilanteissa headless-toteutus voisi toimia, mitä asioita headless-toteutuksissa olisi otettava huomioon ja miten toteutus voidaan tehdä.

2.3 Tutkimusmenetelmät

Tutkimuksessa pyritään vastaamaan laadullisessa tutkimuksessa oleellisiin mitä- ja miten-kysymyksiin. Tarvittaessa tutkimuksessa vastataan tarkentaviin miksi-kysymyksiin, joiden avulla nostetaan kiinnostusta aihetta kohtaan. Headless-toteutuksen teknologioiden (luku 3) tulee käsittelemään toteutuksen hyvät ja huonot puolet, jotka saattavat vaatia tarkentavaa miksi-kysymystä tietyn teknologian osalta, jotta selviää, miksi kyseinen teknologian osa on oleellinen osa headless-toteutusta. Lähteinä pyritään käyttämään mahdollisimman luotettavaa ja tuoretta tietoa, jonka myötä tutkimuksessa välitetty tieto olisi mahdollisimman luotettavaa ja ajantasaista. Tutkimuksessa käsitellään huolellisesti frontend-kehitykseen oleellisia teknologioita, jonka myötä saavutetaan mahdollisimman laaja kuva, miten sovelluksen eri osat vaikuttavat kokonaisuuteen. Mikäli selittäminen vahingoittaa itse sisältöä voidaan kyseinen osuus yksinkertaistaa ja tiivistää. Tämä varmistaa, ettei oleellisia asioita jätetä pois, vaan tuodaan esille ja tiedostetaan niiden olemassaolo. Näihin asioihin saattavat kuulua eri teknologioiden haitat ja hyödyt. (Juhila n.d.)

Tutkimuksessa käytetään laadulliseen tutkimukseen ominaista faktanäkökulmaa, jonka myötä tutkitussa aiheessa keskitytään faktatietoon (Jokinen n.d). Tutkimuksessa faktatieto viittaa työssä käytettäviin lähteisiin, jotka ovat muiden asian tuntevien henkilöiden keräämää ja kirjoittamaa tietoa. Lähteistä kerätään aiheeseen oleellinen tieto, jota käytetään alustamaan ja tukemaan tutkimuksen sisältöä. Lähteiden valinnassa kiinnitetään huomiota siihen, kuka on kirjoittanut, milloin ja onko samankaltaista tietoa saatavilla muualta. Lähteistä tehdään tiivistelmä, jonka pohjalta tieto lisätään opinnäytetyöhön.

Vaikka opinnäytetyö vaikuttaa kehittämistyöltä se vastaa enemmän laadullista tutkimusta missä selvitetään toteutuksesta saatavat hyödyt, haitat ja käytettävät teknologiat nostamatta ennakkoluuloja mitään osa-aluetta tai lopputulosta kohtaan. Aiheesta pyritään tekemään mahdollisimman laaja tutkimus, miten headless-ympäristö voidaan toteuttaa käyttäen muun muassa aiemmin määritettyjä teknologioita. Kyseisen tutkimuksen avulla lukijalle selventyy mikä headless-toteutus on, miten headless-ympäristö voidaan toteuttaa ja missä tapauksissa headless-toteutus voisi olla kannattava toteutustapa.

2.4 Tutkimuskysymykset

Toimeksiantajan tarpeiden mukaisesti opinnäytetyön tavoitteena on toteuttaa toimiva headless-ympäristö, jota voi käyttää pohjana tulevissa headless-projekteissa, jotka käyttävät WordPressiä sisällönhallintajärjestelmänä, GraphQL-kyselykieltä tietojen välittämisessä ja frontendinä frameworkiä, kuten Next.js:ää. Opinnäytetyössä pyritään myös vastaamaan headless-toteutuksessa esiintyviin kysymyksiin:

- Mitkä ovat headless-toteutuksen hyödyt ja haitat? Kysymyksen tarkoituksena on selvittää, mitä hyötyä headless-toteutuksesta on ja minkä takia toteutustapa olisi hyvä ottaa käyttöön.
- Miten ympäristö toteutetaan? Kysymyksellä on tarkoitus ottaa selvää, onko headless-toteutuksessa raajavia tekijöitä, jonka vuoksi toteutusta ei voitaisi tehdä tietyllä teknologialla.

Tutkimuskysymyksiä käsitellään pääasiassa luvun kolme headless-toteutuksen teknologioiden yhteydessä. Luvussa käydään läpi teknologiat, joilla toteutetaan empiirisen osuuden headless-toteutus. Teknologioiden esittelyn yhteydessä selvitetään teknologioista saatavat hyödyt ja haitat headless-toteutuksen yhteydessä.

3 Headless-toteutuksen teknologioita

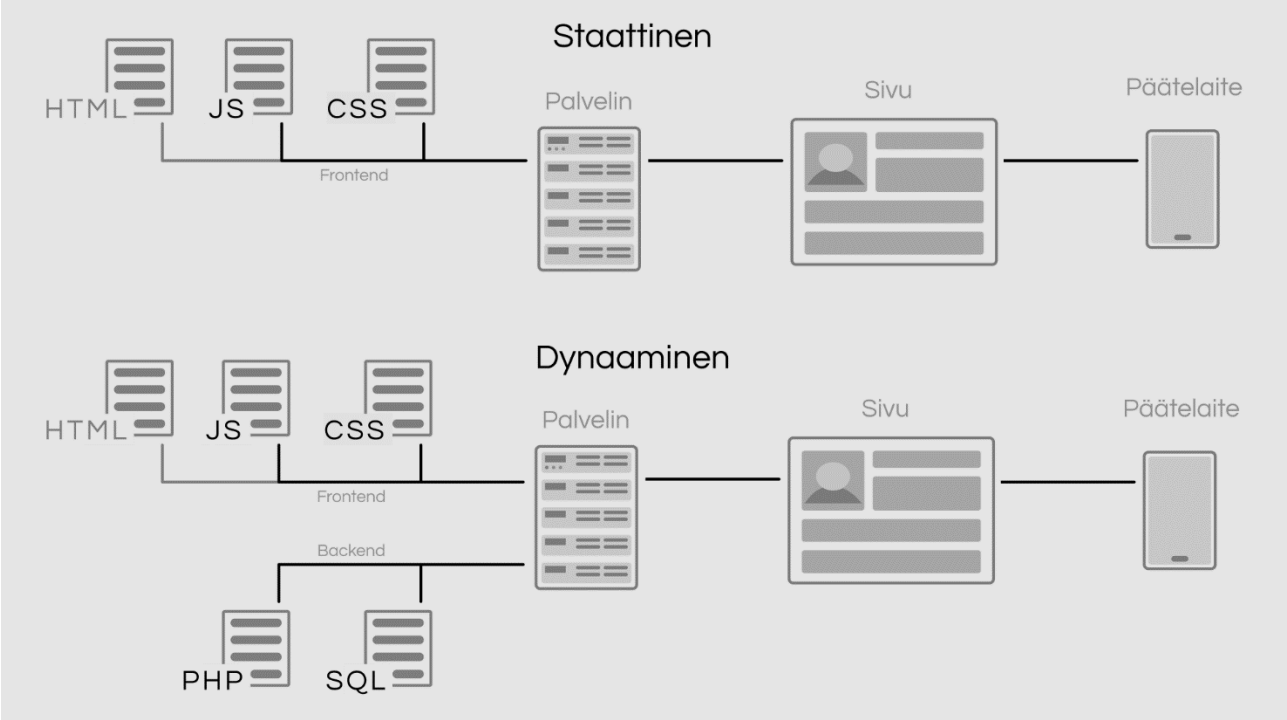
Headless-toteutuksen teknologioiden luku on suunnattu henkilöille, jotka ovat kiinnostuneita headless-toteutuksessa käytettävistä teknologioista. Teknologioiden esittelyn yhteydessä käydään läpi toteutuksen mahdolliset hyödyt ja haitat, jonka jälkeen toteutetaan mahdollinen toteutustapa hyödyntäen luvussa esiteltyjä teknologioita.

3.1 Sisällönhallintajärjestelmä

Sisällönhallintajärjestelmät (eng. Content Management System lyh. CMS) tarjoavat tavan ylläpitää sivustojen sisältöä ilman, että kehittäjien tarvitsee muuttaa sivustojen lähdekoodia. Tällaisia sivustoja voidaan kutsua dynaamisiksi sivustoiksi.

Sisällönhallintajärjestelmiin kuuluu mm. perinteisiin teknologioihin pohjautuvat WordPress ja Drupal sekä uudempiin teknologioihin pohjautuvat Strapi, Contentful ja Netlify. Traditional CMS vs Headless CMS (2021) artikkelin mukaan perinteiset sisällönhallintajärjestelmät sisältävät valmiin sovelluskehitys-ympäristön, joka usein rajoittaa miten ja minkälaisilla teknologioilla kehitystyö voidaan toteuttaa, kun taas uudempiin teknologioihin pohjautuvat sisällönhallintajärjestelmät tarjoavat joustavuutta teknologioiden valinnassa. Headless-sisällönhallintajärjestelmät ovat irrallisia järjestelmiä, jotka tulevat vaatimaan erillisen frontend-ratkaisun (Traditional CMS vs Headless CMS 2021). Headless-sisällönhallintajärjestelmät antavat kehittäjille vapauden valita, miten frontend-toteutus voidaan toteuttaa. Lisää aiheesta headless-toteutuksen luvussa 4.2.

Dynaamisien sivustojen vastakohtana toimii staattiset sivut, jotka eivät sisällä sisällönhallintajärjestelmää, jonka takia ainoa tapa muuttaa sivuston sisältöä on muokata sen lähdekoodia. Vaikka suurin osa sivustoista ovat nykyään dynaamisia se ei tarkoita, etteikö staattisilla sivustoilla olisi omaa roolia nykypäivän internet-arkkitehtuurissa (Goyal n.d). Tekijät, jotka saattavat hyötyä staattisista sivuista sisältää mm. yrityksiä, jotka eivät aktiivisesti päivitä sivustojensa sisältöä, markkinointisivustot ja nettisovellukset (esim. pelit ja sovellusdemot). Aiemmin mainitut tekijät saattavat hyötyä enemmän staattisien sivujen tuomista pienistä ylläpito vaatimuksista ja suhteellisen lyhyistä kehitysajoista (Goyal n.d). Kuviossa 1 on kuvattu staattisensivun, että dynaamisensivun eroja. Huomiona ettei frontend-toteutus välttämättä tarvitse erillistä HTML-tiedostoa, jos se on upotettu JavaScript-tiedostoon.



Kuvio 1. Staattisen ja dynaamisen sivun rakenteet (Static vs Dynamic Website 2021, muokattu)

On hyvä myös erottaa staattiset-sivut generoiduista staattisista-sivuista. Monet nykypäivän sovelluskehitystyökalut, kuten React ja Next.js (luku 3.4) generoivat lähdekoodista lopulta staattisen sivun joko palvelimella tai päätelaitteella. Staattisien sivujen generoinnissa voidaan hakea tietoa erilaisista tietokannoista ja sisällönhallintajärjestelmistä, mikä tekee generoiduista staattisista sivuista todellisuudessa dynaamisia. Tällä toimenpiteellä on omat hyötynsä ja haittansa, joita käsitellään enemmän React ja Next.js-teknologioiden luvussa 3.4.

3.2 WordPress

WordPress on PHP-ohjelmointikielestä pohjautuva sisällönhallintajärjestelmä, joka nousi suosioon 2000-luvun alulla sen helpon käyttöönoton ja ylläpidon ansiosta. WordPress oli 2000-luvun alulla pääsääntöisesti blogisivustoissa käytetty järjestelmä, mikä on ajan saatossa onnistuneesti siirtynyt yleiseen käyttöön. WordPress statistics (2022) selvityksen mukaan n. 40 % verkkosivuista käyttää WordPress-sisällönhallintajärjestelmää.

WordPressin julkaiseman 4.7 päivityksen mukana tuotiin uusi ominaisuus, joka lisäsi WordPressin käyttötapoja. Kyseinen ominaisuus tunnetaan nimellä REST API, jonka avulla WordPress-ympäristön ulkopuolella toimivat sovellukset pystyvät tekemään pyyntöjä kyseiseen WordPress-ympäristöön saadakseen WordPressin sisältämiä tietoja. REST API:n tavoitteena oli alentaa WordPress-ympäristön ulkopuolella toimivien sovelluksien toteutuskynnystä, joihin kuuluu myös headless-toteutukset. (Support 2016.)

Teemat

WordPress-ympäristö koostuu erilaisista komponenteista, joista yksi on teema, joka toimii frontend-ratkaisuna. Ympäristöön voidaan kehittää oma teema, lapsiteema, joka käyttää pohjanaan isäntäteemaa tai viimeisenä voidaan käyttää valmiiksi tehtyä teemaa. Jokaisessa lähestymistavassa on hyvät ja huonot puolensa.

Oman teeman kehittäminen pitää monessa tilanteessa aloittaa tyhjästä, mikä tulee vaatimaan sovelluskehityskokemusta, jonka myötä lopullinen teema tulee vastaamaan sivuston tarpeita. Lapsiteeman kehittäminen ja käyttöönotto saattaa koitua ongelmalliseksi, jos isäntäteema sisältää liian paljon ominaisuuksia tai jos sovelluskehitystaitoinen henkilö ei tunne isäntäteemaa jo entuudestaan. Valmis teema on erittäin helppo ottaa käyttöön, koska se ei vaadi sovelluskehityskokemusta, mutta sivustolle on vaikeampi tuoda haluttuja ominaisuuksia, mikä saattaa vaikuttaa miltä lopullinen sivusto tulee näyttämään.

Lisäosat

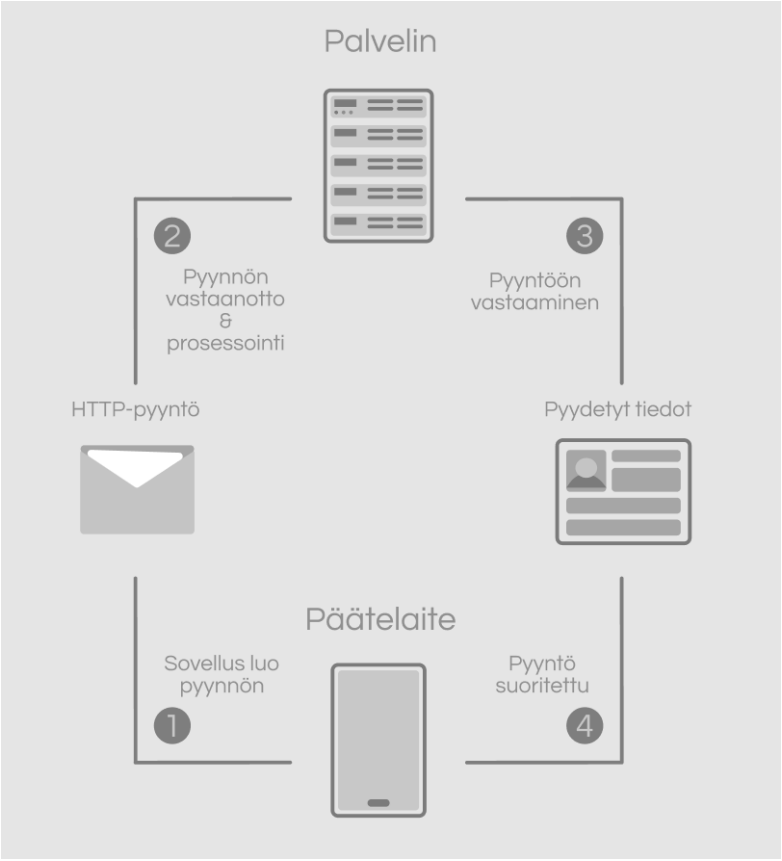
WordPressin yksi suurimpia vahvuuksia on sen laaja lisäosa-kirjasto. Lisäosien lukumäärää pystyy seuraamaan WordPressin verkkosivuilla lisäosat (n.d) sivulta, jonka mukaan helmikuussa 2022 ilmaisien lisäosien määrä oli 59 470. Lisäosien avulla voidaan tuoda sovellukseen lisätoiminnallisuuksia lisäämättä niitä suoraan teemaan, mikä helpottaa kummankin teeman, että lisäosan ylläpitoa.

3.3 Tiedonsiirto sovelluksissa

Sovelluksissa voidaan monella tavalla hakea tietoa erilaisista lähteistä, minkä vuoksi on hyvä tietää erilaisien tapojen vahvuuksia ja heikkouksia. Teknologioiden rajojen ymmärtäminen saattaa parantaa lopullisen sovelluksen ylläpitoa ja käyttökokemusta.

REST-API

Monet sovellukset saavat tietonsa käyttäen REST API:a, joka on kevyt ja nopea tapa tuoda tietoja tietokannasta sovellukseen. Yhden maailman johtavimman teknologiayhtiön IBM:n mukaan REST API on suosituin mikropalvelu. REST API:n toiminnallisuus perustuu HTTP-pyyntöihin, joita tekemällä sovellus lähettää palvelimelle pyynnön, johon REST API vastaa palauttamalla JSON-muodossa pyydetyn tiedon. Kuviossa 2 on kuvattu HTTP-pyynnön rakennetta, jossa pyynnön aloittaja on päätelaite, joka myös päättää pyynnön ottamalla vastaan palvelimelta tulevan vastauksen. (REST APIs 2021.)



Kuvio 2. HTTP-pyynnön rakenne kuvattu vaiheittain (REST APIs 2021, muokattu)

GraphQL

WordPress-lisäosakirjasto tarjoaa monia hyödyllisiä lisäosia, joista yksi on WPGraphQL, joka mahdollistaa GraphQL:n käytön WordPress-ympäristössä. Rascian (2021) mukaan GraphQL ja REST API täyttävät saman tehtävän eli hakevat tietoja tietokannasta ja palauttavat sen JSON-muodossa sovellukseen. Verrattuna perinteiseen REST API:in, GraphQL sisältää hyödyllisiä ominaisuuksia, jotka saattavat helpottaa kehitystyötä tietokantojen parissa.

GraphQL on Facebookin vuoden 2015 julkaisema kyselykieli, jonka tarkoituksena on tarjota uusi tapa hakea tietoja tietokannoista. GraphQL-kyselykieli on suunniteltu toimimaan yhden päätepisteen varassa käyttäen HTTP-kyselyitä, mikä parantaa suorituskykyä ja lisää joustavuutta. (Ieva 2021.)

Vaikka GraphQL on suunniteltu toimimaan yhden päätepisteen varassa, sen yhtenä etuna on hakea tietoja useasta päätepisteestä yhdellä pyynnöllä, kun taas REST API vaatii erillisen pyynnön jokaiseen päätepisteeseen saadakseen saman tuloksen. REST API:en kasvaessa vanhojen tietojen pois jättäminen ja uusien lisääminen saattaa aiheuttaa ongelmatilanteita, jotka saattavat vaatia uusien REST API versioiden julkaisemista tulevaisuudessa. GraphQL välttää kyseisen ongelman tekemällä tietojen muuttamisesta suhteellisen helppoa, näyttämällä mitä tietoja voidaan palauttaa GraphQL:n sisältämän käyttöliittymän avulla ja palauttamalla täsmälleen sen tiedon mitä sovellus pyytää. Nämä ominaisuudet tekevät GraphQL-kyselykielestä muutosystävällisen, johon voidaan lisätä uusia arvoja rikkomatta sovelluksia, jotka hakevat tiedot kyseisestä päätepisteestä. (Ieva 2021.)

GraphQL saattaa vaikuttaa hyvinkin lupaavalta vaihtoehdolta, mutta siinäkin on omat ongelmansa. Gopalakrishnan (2021) mukaan ongelmia saattavat aiheuttaa mm. virheviestit, turvallisuus, N+1-ongelmat ja suorituskykyongelmat.

GraphQL palauttaa aina HTTP-tilakoodin 200, joka on positiivinen vastaus ja tarkoittaa ettei pyynnön lähettämisessä tai vastaanotossa havaittu ongelmia, mikä ei helpota virheen paikantamista, kun välitetyt tiedot ovat virheellisiä (Gopalakrishnan 2021). Turvallisuuden näkökulmasta Gopalakrishnan (2021) väittää, ettei kyselyiden teko tarpeen mukaan ole aina hyvä ratkaisu varsinkaan

monimutkaisien kyselyiden osalta, koska kyselyt saattavat ylikuormittaa sovelluksen palvelimen, mikä altistaa sovelluksen palvelunestohyökkäykselle (eng. Denial of Service attack lyh. DoS). N+1-ongelma viittaa ongelmaan, jossa palvelimen pitää hakea useamman kerran tietoa, koska aiempi pyyntö ei sisältänyt kaikkea sovelluksen tarvitsemaa tietoa (Gupta 2021).

Viimeinen GraphQL:n ongelmatilanne esimerkki liittyy suorituskykyyn, jota aiemmin pidettiin GraphQL:n yhtenä vahvuutena, mikä saattaa vaikuttaa tällä hetkellä ristiriitaiselta. Sovelluskehityksessä saatetaan päätyä lopputulokseen, joka on raskaampi kuin aiemmin suunniteltu lopputulos, mikä saattaa hidastaa sovellusta ja vahingoittaa käyttäjäkokemusta. Gopalakrishnan (2021) mukaan GraphQL-kyselykielessä suorituskyky ongelmaan saatetaan päätyä, kun sovellus sisältää monia monimutkikkaita ja raskaita kyselyitä, jonka vuoksi GraphQL yksin ei välttämättä riitä prosessoimaan niitä. Yhtenä ratkaisuna Gopalakrishnan (2021) suosittelee käyttämään natiivia GraphQL-tietokantaa, joka pystyy käsittelemään monimutkaisiakin kyselyitä millisekuntien vasteajalla.

GraphQL ei ole kaikissa tapauksissa parempi lähestymistapa kuin perinteinen REST API, joten on hyvä tehdä taustatyötä projektin näkökulmasta ja tehdä lopullinen päätös sen kautta. GraphQL-kyselykieli on kuitenkin hyvä työkalu, joka sisältää hyödyllisiä ominaisuuksia, joita ei esiinny REST API:ssa.

3.4 Next.js & React-kirjasto

Next.js on Vercel-yhtiön kehittämä React-kirjastosta pohjautuva framework, joka on myös Reactin kehitystiimin suosittelema työkalu (Create a New React App n.d). Koska Next.js-framework pohjautuu React-kirjastoon, pystyvät entuudestaan Reactia osaavat henkilöt helposti siirtymään Next.js-ympäristöön.

Seuraavaksi käydään läpi, miten Next.js ja muut Reactista pohjautuvat frameworkit saivat alkunsa. Ennen Reactista pohjautuvia frameworkejä kehittäjillä oli mahdollisuus kehittää sovelluksia Facebookin kehittämällä React-kirjastolla, joka helpotti kehitystyötä ja antoi kehittäjille uusia tapoja ratkoa ongelmia. Kun Reactin suosio nousi, niin nousi myös raskaiden nettisivujen määrä. Raskaiden nettisivujen kasvu React-sovelluksien osalta johtui työkalusta, josta React pohjautuu eli JavaScriptistä. Selaimet, laitteet ja hakukoneet on alun perin suunniteltu tukemaan HTML-sivuja, jotka ovat staattisia eivätkä sisällä raskasta logiikkaa. Reactin suosion noustessa yhä useampi sivu

rakentui suurista JavaScript paketeista, joiden prosessointi vie enemmän aikaa ja resursseja kuin HTML-sivujen prosessointi. Tämä teknologinen muutos näkyi negatiivisesti varsinkin hakukoneissa. (Mohan 2020.)

Hakukoneet on suunniteltu lukemaan tietoa HTML-sivuista, minkä vuoksi hakukoneet eivät pysty kertomaan minkälaista sisältöä löytyy JavaScript-paketeista koostuvilta React-sivuilta, mikä vahingoittaa sivujen näkyvyyttä hakutuloksissa. Kehittäjät alkoivat kehittämään erilaisia ratkaisuja parantamaan hakukonenäkyvyyttä, joista yksi oli Reactilla tehtyjen sivujen prosessointi palvelimella, jonka jälkeen palvelin lähettäisi JavaScriptistä prosessoidun HTML-sivun. React-sovelluksen prosessointi palvelimella siirtäisi päätelaitteella suoritetun vaiheen laitteelle, joka on kykeneväinen prosessoimaan React-sovelluksen suhteellisen lyhyessä ajassa. Ratkaisu on todettu toimivaksi, mikä valitettavasti vie liikaa kallisarvoista aikaa sovelluskehityksestä. Tämä on yksi syy miksi monet kehittäjät ovat siirtyneet erilaisiin React pohjaisiin frameworkeihin, kuten Next.js:ään, jotka generoivat Reactin JavaScript-koodista HTML-sivuja eli staattisia sivuja ja tuovat muita kehitystyötä tukevia ominaisuuksia. (Mohan 2020.)

Googlen tekemien hakukone indeksointimuutoksien myötä Googlen hakukone on kykeneväinen tulkitsemaan myös sivustoja, jotka pitää prosessoida päätelaitteella. Tämä ei ole kuitenkaan veden pitävä uudistus, jonka vuoksi hakutulosongelmia saattaa ilmaantua vieläkin React-sovelluksissa vuonna 2022. (Baranowski 2021.)

Kiteytettynä Next.js on yksi monista ratkaisuista Reactin tuottamiin ongelmiin, mikä sisältää monia paketteja ja ominaisuuksia, jotka tekevät kehityksen aloittamisesta mahdollisimman vaivatonta. Lisää Next.js:stä luvussa 4.3.

4 Headless-toteutus

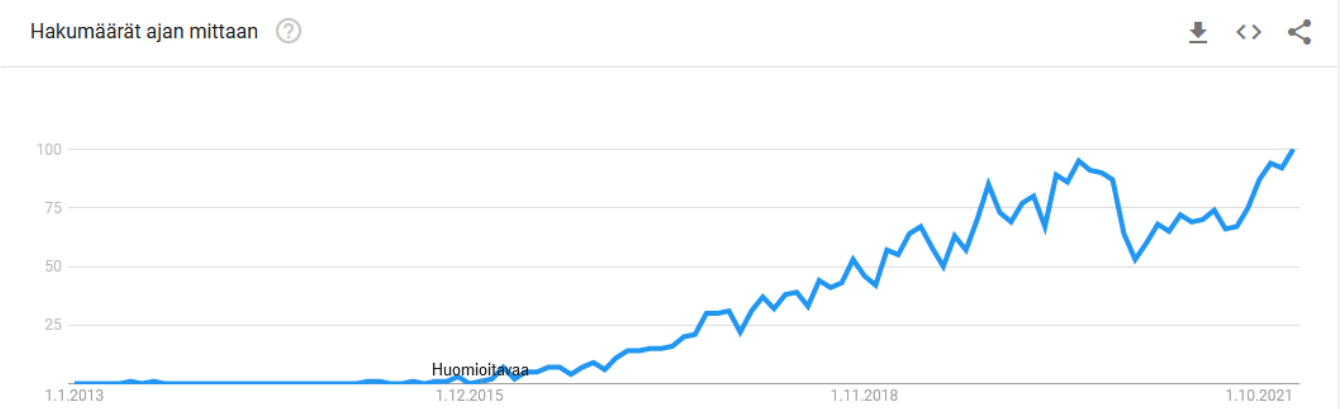
Headless-toteutus on toteutustapa, jossa erotetaan backend frontendistä, jolla Angererin (n.d.) mukaan saavutetaan sovelluksen itsenäisyys, jolla voidaan käytännössä tarkoittaa kehittäjien vapautta päättää millä teknologioilla projekti tullaan toteuttamaan. Singh (2020) mukaan tämä avaa myös uusia mahdollisuuksia, miten sovellus voidaan suunnitella, kun sovelluksen frontend-arkkitehtuuri on täysin avoin sovelluksen kehittäjille.

WordPressin perinteisessä toteutustavassa ympäristö koostuu valmiiksi kootusta kehitysympäristöstä, jossa voidaan muokata sovelluksen ulkonäköä kehittämällä uusi teema käyttäen PHP-, CSS- ja JavaScript-ohjelmointikieliä. WordPress-ympäristössä kyseisiä teknologioita voidaan hyödyntää myös lisäosien kehityksessä, joilla voidaan muokata sovellusta lisäämättä uutta ominaisuutta teeman kautta.

Verrattuna headless-toteutukseen WordPress sulkee pois monia uusia teknologioita, mikä saattaa rajata mitä ominaisuuksia sovellukseen voidaan lisätä ja kuinka helposti. Headless-ympäristö ei vaadi tiettyjä taitoja, kuten PHP-ohjelmointikielen tuntemusta, vaan frontend saatetaan toteuttaa kokonaan teknologioilla, jotka ovat kehittäjille entuudestaan tuttuja, kuten JavaScriptillä, käyttäen erilaisia frameworkkejä, kuten Next.js, Vue tai Angular.

4.1 Headless-toteutuksen suosio

Kuviossa 3 on kuvattu Google Trends kaavio, jonka mukaan ”headless cms”-hakutuloksen suosio on noussut tasaisella tahdilla 2015 vuodesta lähtien. Kasvun alkuperää on vaikea paikantaa, mutta osa tätä kasvua saattavat olla 2013 ja 2016 vuosien välillä julkaistut frameworkit, joita usein käytetään headless-toteutuksissa.



Kuvio 3. "Headless CMS"-hakutuloksen suosio (Google Trends 2022)

Kuten monissa teknologisissa edistymisissä monet yhtiöt haluavat oman tuotteen markkinoille ja framework kehitys ei ole poikkeus sillä Facebook julkaisi 2013 heidän uuden frameworkin Reactin,

seuraavana vuonna Googlen entinen työntekijä Evan You julkaisi frameworkin Vue.js:n ja 2016 Google julkaisi heidän frameworkin Angularin. (Mohan 2020.)

Mohanin (2020) artikkelin pohjalta voidaan myös tehdä johtopäätös, että headless-toteutuksien nousevan suosion toisena tekijänä saattaa olla räätälöityjen sivustojen tarpeen kasvu ja headless-toteutuksien kynnyksen lasku aiemmin mainittujen työkalujen julkaisun myötä.

Headless-toteutukset ovat joustavampia kuin perinteisellä tavalla toteutetut sovellukset, mikä on hyvin voinut olla myös aktiivinen tekijä headless-toteutuksien kysynnän nousuun. Niżyńskin (2020) mukaan headless-toteutuksien joustavuus mahdollistaa myös lyhyemmän reaktioajan. Reaktioaika vaatimukset tulevat olemaan eriäviä sovelluksesta riippuen esim. verkkokaupat tulevat vaatimaan lyhyen reaktioajan minimoidakseen mahdollisien ongelmatilanteiden aiheuttamat tappiot.

4.2 Headless-toteutus WordPressissä

WordPress sisältää sovelluksen frontendin teeman muodossa ja backendin tietokannan sekä sisältöhallintajärjestelmän muodossa, joten WordPressistä pitää erotella frontend backendistä, jotta saadaan termin mukainen headless-ympäristö.

Ennen kehitystyön aloittamista olisi hyvä määritellä käytettävissä olevat teknologiat ja pystyttää toimiva kehitysympäristö, jolla on pääsy päätepisteisiin. Sovelluksen hyvin suoritettu suunnittelu- ja pystytysvaihe saattavat helpottaa sovelluskehityksen etenemistä.

WordPress-ympäristössä headless-toteutus ei juuri vaadi muuta kuin tietopäätteiden määrittämistä GraphQL:n tai REST API:n avulla. Mikäli WordPressissä on käytössä WYSIWYG-tekstieditori Morriksen (2022) mukaan se ei välttämättä saavuta samaa lopputulosta headless-toteutuksessa, minkä takia WordPress saattaa vaatia ACF-lisäosan (kok. Advanced Custom Fields), jonka avulla voidaan luoda uusia tekstikenttiä, joilla voidaan saavuttaa sama lopputulos kuin WYSIWYG-tekstieditorilla. Frontend tulee vaatimaan enemmän kehitystyötä headless-toteutuksessa, jos halutaan saavuttaa samat toiminnallisuudet kuin WordPress-toteutuksessa. WordPress sisältää muun muassa seuraavia toiminnallisuuksia: kommentointi, arkistosivu(t), navigaatiovalikon luominen ilman koodia ja sivujen luominen reaaliajassa.

WYSIWYG-tekstinkäsittelyohjelma tarkoittaa ”What you see is what you get”-tekstinkäsittelyohjelmaa, joka kääntyy ”Se mitä näet on mitä saat”. Tekstinkäsittelyohjelman nimi on hyvin kuvaava, mikä käytännössä viittaa, että WYSIWYG- tekstinkäsittelyohjelma antaa käyttäjilleen intuitiivisen tavan lisätä sisältöä ja muokata miten se esitellään sivulla. WordPress-ympäristössä kyseinen tekstinkäsittelyohjelma tunnetaan nykyään Gutenberg nimellä, mikä tarjoaa visuaalisen tavan rakentaa sivun ulkonäköä käyttäen komponentteja, joilla on erilaisia ominaisuuksia. Ominaisuuksien muuttaminen vaikuttaa miten komponentin sisältämä tieto esitellään. (Huss 2022.)

WordPressin headless-ympäristössä tietojen syöttäminen ja muokkaaminen sujuu normaalisti WordPressillä, joka kuitenkin saattaa vaatia lisäosan, joka mahdollistaa uusien tekstikenttien lisäämisen. Tietojen haku sujuu helposti REST API:n tai GraphQL:n avulla WordPressistä, mikä lähetettään Next.js-sovellukseen. Tarvittaessa sovellukseen voidaan tuoda samat toiminnallisuudet mitkä löytyvät WordPressistä, mutta ne on kehitettävä itse.

4.3 Next.js headless-toteutuksessa

Headless WordPress-toteutuksessa Next.js korvaa WordPressistä pois jäävän frontendin. Toisin kuin WordPressissä Next.js-ratkaisua ei voida muokata jälkeenpäin lisäämällä lisäosia, joilla voitaisiin tuoda lisäominaisuuksia ilman sovelluskehittäjiä. Sivusto tulee tarvitsemaan jatkokehitystä varten sovelluskehittäjiä, jotka suunnittelevat ja tuovat haluttuja ominaisuuksia sovellukseen. (Morris 2022.)

Headless-toteutuksen sovelluskehityksessä esiintyvien vaatimuksien vuoksi headless-ratkaisu ei välttämättä kuulosta houkuttavalta tekijöille, jotka haluavat vain toimivan WordPress-nettisivuratkaisun. Tämän vuoksi on hyvä tiedostaa, ettei headless-toteutus tule välttämättä tarvitsemaan jatkokehitystä, jos omistaja ja asiakkaat ovat tyytyväisiä headless-toteutukseen. Onnistuneen toteutuksen saavuttaminen tulee vaatimaan hyvän suunnitelman, joka edellyttää mm. kilpailijoiden, asiakaskunnan ja asiakasyrityksen tutkimista.

Kehittäjäystävällinen ympäristö voi tarkoittaa monelle eri asioita ja eri frameworkeissä eri ominaisuuksia. Next.js:n ja tämän opinnäytetyön tapauksessa kehittäjäystävällinen ympäristö tarkoittaa ympäristöä, joka sisältää ominaisuuksia, jotka nopeuttavat kehitystyötä. Vercel-yhtiön ylläpitämän

Next.js Getting Started (n.d) dokumentaation mukaan Next.js sisältää mm. palvelin puoleisen renderöinnin (eng. server rendering), sisään rakennetun CSS tuen, fiksu niputuksen (eng. smart bundling), ympäristö muuttujat, tietojen haun (eng. data fetching) ja reitin esihaun (eng. route prefetching).

Headless-toteutuksen frontendin osana Next.js tarjoaa kehittäjäystävällisen ympäristön, johon pystytään tarvittaessa lisäämään kolmannen osapuolen palveluja tuomaan joitakin ominaisuuksia, joihin ei ole oleellista kehittää omia ratkaisuja (Singh 2020). Tämä estää resurssien upottamisen yleisien ominaisuuksien suunnitteluun ja ylläpitoon, joissa voisi käyttää muiden sovelluskehittäjien ylläpidettäviä ratkaisuja, jotka sisältävät omat tukikanavansa, joita voidaan hyödyntää ongelmien ilmentyessä mm. implementaatiossa tai tulevaisuuden päivityksissä. Kyseisiin kolmannen osapuolen palveluihin voi kuulua maksutapahtuma API, statistiikkaa seuraava API tai kirjasto, joka mahdollistaa tuotteiden 3D-mallien tuonnin tuotesivuille.

Kiteytettynä kun WordPressistä tehdään headless-toteutus frontend erotetaan backendistä eli aiemmin toiminut WordPress-teema pitää korvata jollain frontend ratkaisulla esimerkiksi framework-toteutuksella kuten sovelluksella, joka on tehty Next.js-frameworkillä.

5 Headless-toteutukset maailmalla

Headless-toteutukset ovat olleet jo useamman vuoden laajassa käytössä maailmalla, mutta epäonnistuneet saavuttamaan laajaa käyttöä Suomessa. Joten tässä tapauksessa on hyvä lähteä katsastamaan, miten uusia teknologioita on käytetty maailmalla ja oppia muiden virheistä ja onnistumisista. Tähän tarkoitukseen case-tutkimukset tarjoavat kattavan ja joskus hyvin avartavan katsauksen projektien tavoitteisiin ja mitä lopputuloksilla saavutettiin. Huomiona etteivät tässä luvussa esitetyt case-tutkimukset käytä aiemmin käsiteltyjä frontend tai backend teknologioita. Luvussa käydään läpi OneBlade Increases Refill Subscription Rate By 579% (n.d) ja Umbraco helped Skoringen tie their laces (n.d) nimisiä case-tutkimuksia.

5.1 OneBlade headless-toteutus

OneBlade on USA:ssa toimiva partaterien tuottaja ja jälleenmyyjä, jonka perustaja halusi tuoda italialaisen parranajo kokemuksen USA:han ylellisillä ja laadukkailla tuotteilla. Kuten vastaavat parranajotuotteet OneBlade toimii kuukausitilausmallilla.

Alun perin OneBlade-sivusto oli toteutettu räätälöidyllä frontend-ratkaisulla, joka oli yrityksen kasvun myötä paisunut erittäin raskaaksi. Asiaa ei auttanut monimutkainen tilausprosessi, joka heikensi käyttäjäkokemusta entisestään.

OneBlade Avex-yhtiön kanssa päätyivät Shogun headless-ratkaisuun, joka on sähköiseen kaupankäyntiin erikoistunut alusta. Shogun tarjoaa valmiin frontend headless-ympäristön nettisivumalleineen, joita voidaan helposti muokata asiakkaan tarpeiden mukaisesti. OneBlade:n vaatimuksiin kuului parempi käyttäjäkokemus ja ratkaisu, joka käyttää olemassa olevaa Shopify backend-ratkaisua.

Käyttäjäkokemusta heikentävinä tekijöinä olivat 12 sekunnin lataus ajat ja useamman sivun läpi kestävä tilausprosessi. OneBlade brandi on erittäin visuaalinen, jonka myötä vanha sivusto sisälsi paljon kuvia ja animaatioefektejä, jotka vaikuttivat myös mille sivulle OneBlade pystyisi lisäämään tuotevideoita. OneBlade oli myös tiedostanut heidän tilausprosessinsa olevan liian pitkä, jonka vuoksi tilausprosessia haluttiin lyhentää mahdollisimman lähelle yhtä klikkausta.

Asiakkaan käyttäjäkokemuksen lisäksi haluttiin helpottaa markkinointihenkilöstön ja sovelluskehittäjien työtä lisäämällä koodittoman sisällönhallintaympäristön, jota pystyisi myös markkinointi puolen henkilö käyttämään. Backend lisäominaisuuksien lisäksi kehittäjät pystyvät lisäämään headless-toteutuksessa kolmannen osapuolen työkaluja, kuten ReCharge Payments, jolla käsitellään uusiutuvat tilaukset.

Headless-toteutuksen käyttöönoton jälkeen OneBlade näki positiivisia tuloksia jo ensimmäisen kuukauden jälkeen. OneBlade:n statistiikan mukaan sivuston lataus nopeus oli laskenut 1-2 sekuntiin ja aktiivisien jatkuvien tilauksien määrä oli kasvanut melkein seitsenkertaisesti ensimmäisen kolmen kuukauden aikana.

5.2 Skoringen headless-toteutus

Skoringen on suurin Tanskassa ja Norjassa toimiva jalkine kauppaketju. Yhtiön tavoitteena headless-toteutuksessa oli pysyä kilpailukykyisenä, mikä ei ollut mahdollista vanhentuneilla nettisivuilla. Suurimpana ongelmana vanhoissa sivuissa oli, etteivät ne pystyneet välittämään tietoja kivijalkakauppojen inventaarioista tai tilauksista eli nettisivut olivat täysin irtaantuneita itse kaupoista, minkä takia paras kokemus oli asiakkaalla, joka kävi asioimassa paikan päällä.

Skoringen päämääränä oli saavuttaa headless-toteutus, joka mahdollistaisi sisällön lisäämisen ja muokkauksen helposti, nopeasti, missä vain ja milloin vain. Tietojen pitäisi myös päivittyä reaaliajassa, jotta asiakaskokemus olisi mahdollisimman lähellä mitä Skoringen haluaisi sen olevan.

Skoringen headless-toteutuksessa nopeus on huomion keskipisteenä, jonka takia sivusto ei lataa jokaista sivua erikseen vaan hakee halutut tiedot mikropalveluiden avulla. Tällaista sivustoa voidaan kutsua yksisivuiseksi sovellukseksi (eng. Single-Page Application lyh. SPA), joka nimensä mukaisesti on yksi sivuinen sovellus, johon tuodaan sisältöä reaktiivisesti käyttäjän painamien linkkien tai painikkeiden kautta.

Umbraco tarjoaa sovellukselle kevyen sisällönhallintajärjestelmän, joka vapauttaa resursseja ja siirtää kaikki tarvittavat tiedostot pilvipalvelimelle, jonka ansiosta yhtiöllä on pääsy sivustoihinsa, milloin vain ja missä vain. Tämä saattaa olla mullistava muutos yhtiölle, jonka sivustot ovat olleet vanhentuneita jo pitkään eivätkä ole välttämättä sisältäneet luotettavaa sisällönhallintajärjestelmää aiemmin. Laajan ja moni kielisen asiakaskunnan vuoksi sivuston on pystyttävä räätälöimään sisältöä asiakkaan mieltymyksien, kielen ja käyttäjähistorian mukaan, joka parantaa käyttäjäkokemusta ennestään.

Skoringen headless-toteutus tuo joustavuutta sisällöntuotannon sekä kehittäjien näkökulmasta, joka helpottaa sivuston ylläpitoa. Käyttäjien arvostelut uusista sivuista ovat olleet erinomaiset, mikä viestittää selkeästi, että Umbraco headless-ratkaisu on ratkaissut Skoringen online ja kivijalkakauppojen välisen ongelman. Sivusto on tuonut lisää positiivista liikennettä niin nettisivuilla, kuin myös kivijalkakaupoissa, joissa on nähty noin 50 prosentin kasvu.

5.3 Yhteenveto esimerkeistä

Luvussa käsiteltyjä case-tutkimuksia on hyvä käydä kyseenalaistamalla läpi. Näissä tapauksissa case-tutkimuksilla esitellään mihin case-tutkimuksen tekevän yhtiön teknologia on kykenevä ja millaisissa tapauksissa se saattaa ratkaista ongelmia. Case-tutkimuksien perusteella voidaan päätellä oikeiden sovelluksien kautta minkälaisissa tilanteissa headless-toteutus olisi hyvä lähestymistapa.

Skoringen kenkäkaupan vanhentuneet nettisivut eivät kyenneet näyttämään kivijalkakauppojen valikoimaa reaaliajassa eikä myöskään välittämään nettisivujen kautta tehtyjä tilauksia kivijalkakauppoihin, jonka takia asiakkaat eivät kyenneet noutamaan tilaustaan paikan päältä. Vanhat sivut estivät myös uusien ominaisuuksien lisäämisen, mikä vahingoitti yhtiön kilpailukykyä markkinoilla.

OneBlade ei ollut saavuttanut odotuksiaan ja oli tiedostanut vanhoilla sivustoillaan esiintyviä ongelmia, joihin kuului pitkät latausajat ja pitkän tilausprosessin, jotka tekivät käyttäjäkokemuksesta turhauttavan. OneBlade halusi säilyttää visuaalisen brändäyksen, joka oli osa syyllinen aiempien sivujen heikkoihin latausaikoihin. Kehitystiimin piti myös rakentaa uudet sivut huomioiden, että aiemmin käytetty backend tulee toimia uusissa sivuissa. Vaatimuksien myötä päädyttiin myös valitsemaan muut teknologiat, jotka tukisivat OneBlade:n aiempaa backend-ratkaisua ja visuaalista brändäystä.

Kummatkin yhtiöt löysivät heille sopivan ratkaisun hyödyntämällä headless-toteutusta, jonka ansiosta sivustoille pystyttiin lisäämään halutut ominaisuudet. Ominaisuuksiin kuului mm. tuote ehdotuksia käyttäjähistorian perusteella, nopeat latausajat sisällön raskaasta luonteesta huolimatta ja parannettu statistiikka seuranta, jota voidaan hyödyntää sivuston kehityksessä.

Case-tutkimukset esittelivät myös, kuinka monipuolisia ratkaisuja headless-toteutuksella voidaan saada aikaiseksi. Kummassakin projektissa tehtiin taustatyötä, jonka perusteella rajattiin mahdolliset työvälineet, joilla sivusto tulisi tehdä. Taustatyön ansiosta sivuston ei tarvinnut sopia mihinkään olemassa olevaan muottiin vaan kehitystiimit pystyivät valitsemaan työvälineet ja teknologiat, joilla saavutettaisiin halutut lopputulokset.

6 Headless-sovelluspohjan kehittäminen

Headless-sovelluksen kehittäminen voidaan aloittaa mistä tahansa osasta. Esimerkiksi frontend-toteutus voidaan suunnitella ja tehdä täysin valmiiksi, jonka jälkeen se voidaan yhdistää päätepisteeseen**,** jonka kautta sovellus tulee saamaan dynaamista sisältöä. Kehitystyössä ei kuitenkaan hyödynnetä tällaista toteutustapaa vaan työn jokainen osa-alue luodaan heti alussa ja varmistetaan, että ympäristö toimii kokonaisuudessaan ennen varsinaisen sovelluksen kehitystä.

6.1 Ympäristöjen alustaminen

Kehitystyössä käsitellään Next.js & WordPress headless-ympäristön pystytystä, koska aihe käsittelee astetta kehittyneempää kehitystyön aihetta, työssä oletetaan jonkinlaista tuttavuutta aiemmin käsitellyistä teknologioista. Käytännössä tämä tarkoittaa, ettei kehitystyössä tulla käymään läpi Next.js- tai WordPress-ympäristöjen asennusta. Tarvittaessa Next.js:n sivuilta löytyy Vercelin ylläpitämä ”Getting Started”-dokumentaatio. WordPress tarjoaa myös oman ”Getting Started”-dokumentaation, josta löytyy asennusohjeet ”How to install WordPress”-artikkelista.

WordPress

WordPress-asennuksen jälkeen sisällönhallintajärjestelmään voi syöttää tietoja, joita Next.js tulee käyttämään. Tämä ei ole kuitenkaan välttämätöntä, koska WordPress sisältää jo jonkin verran täyttösisältöä sivujen, artikkeleiden ja kommenttien muodossa, joita voidaan tarkastella WordPress-teeman kautta.

Sisällön lisäksi WordPress tulee tarvitsemaan lisäosia, jotta tietojen vienti ja tiettyjen tietojen lisäys olisi mahdollista. Aiemmin käsitelty GraphQL-lisäosa löytyy WordPressin lisäosakirjastosta nimellä ”WP GraphQL”, jonka on julkaissut WPGraphQL niminen tekijä. On hyvä varmistaa latauksen jälkeen, että lisäosa on myös aktivoitu.

WPGraphQL-lisäosa lisää oman valikon WordPressin admin-näkymän valikkoon, josta päästään mm. asetuksiin, joiden kautta voidaan tarvittaessa esim. rajoittaa kuinka laajoja kyselyitä GraphQL:n kautta voidaan tehdä. ”GraphiQL IDE”-näkymän kautta voidaan tehdä testipyyntöjä, mikä tulee olemaan hyödyllinen ongelmatilanteiden ratkaisussa. Asetuksista löytyvä ”GraphQL

Endpoint”-osoite on hyvä ottaa talteen, koska se mahdollistaa ulkopuolisten sovelluksien pääsyn WordPressin sisältäviin tietoihin eli kyseinen osoite tullaan lisäämään myöhemmin Next.js-sovellukseen. (Taxonomies n.d)

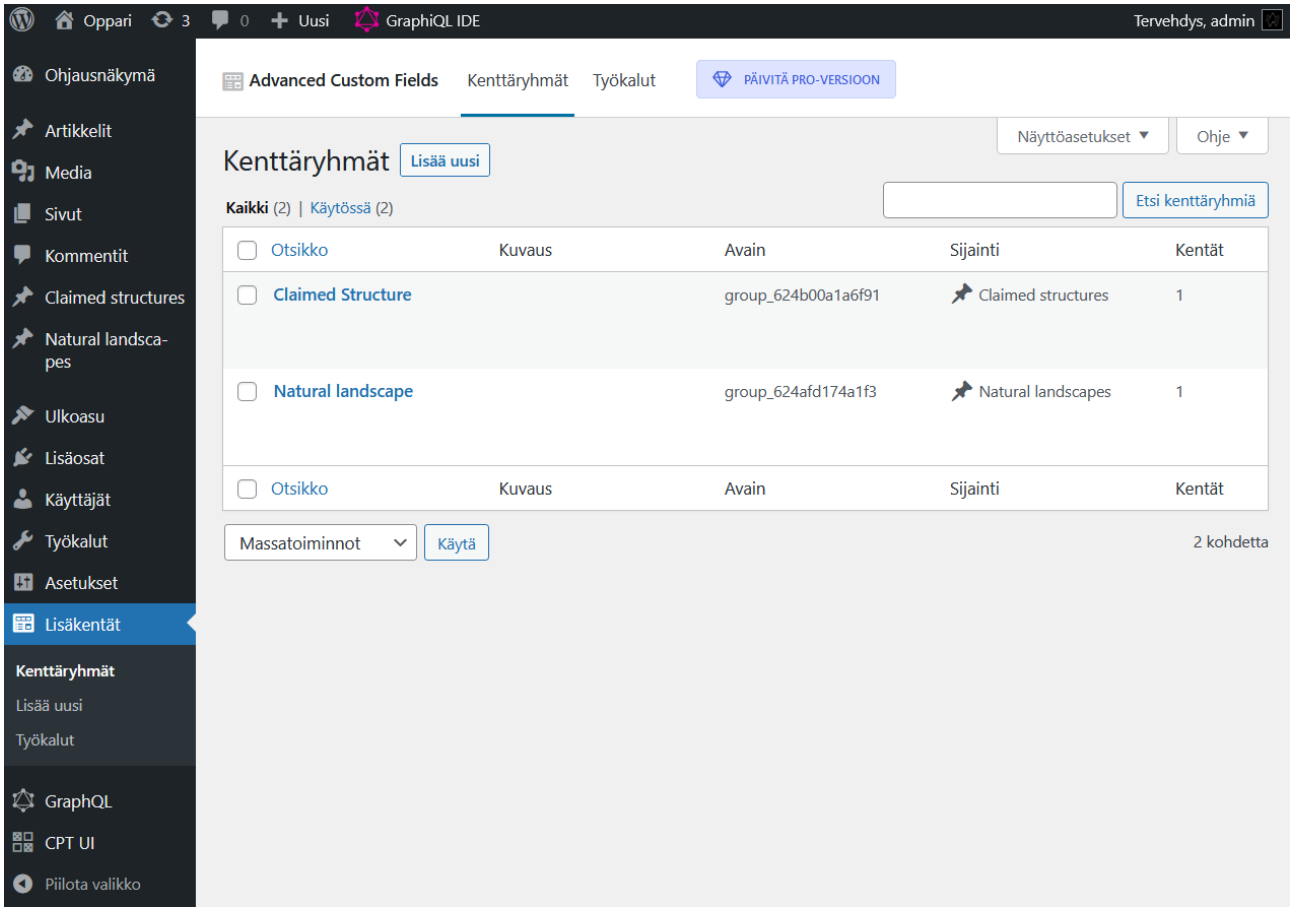
Seuraavana lisätään lisäosa, jonka kautta voidaan luoda artikkelityyppejä (eng. Custom Post Type lyh. CPT) ja taksonomioita (eng. taxonomies) artikkeleihin. Taxonomies (n.d) WordPress-artikkelin mukaan CPT:n ja taksonomian avulla voidaan hallinnoida artikkeleita paremmin erottelemalla ne eri tyyppeihin esim. mielipidekirjoituksiin ja artikkeleihin. Tämän mahdollistaa ”Custom Post Type UI”-lisäosa, joka on WebDevStudios nimisen tekijän julkaisema. Huomiona, että CPT:tä ja taksonomioita voidaan myös luoda itse oman lisäosan tai teeman kautta.

Viimeisenä lisäosana WordPress saattaa tarvita lisäosan, jonka avulla voidaan luoda uusia kenttiä sivuihin ja artikkeleihin. Tämä tulee helpottamaan sisällön hakemista frontend-toteutuksen puolella. Lisäosa ei ole kuitenkaan oleellinen jokaisessa toteutuksessa, jonka myötä lisäosan voi jättää lataamatta. Työssä käytettävä lisäosa on ”Advanced Custom Fields” (lyh. ACF), joka on ”Delicious Brains” nimisen tekijän julkaisema. ACF-lisäosa luo ”lisäkentät”-valikon WordPressin admin-valikkoon, jonka kautta voidaan luoda uusia kenttäryhmiä. Uuden ryhmän lisääminen on suhteellisen suoraviivaista, mutta lisäosa saattaa vaatia jonkin verran perehtymistä. Onneksi ACF-lisäosan kehittäjät tarjoavat lisäosalle hyvin ylläpidettyä dokumentaatiota (Documentation n.d).

Huomiona, että ACF-lisäosassa tulee pistää ”Näytä REST API:ssa”-asetus päälle kenttien luonnin yhteydessä ”Asetukset”-väliotsikon alta, jos ympäristössä käytetään REST APIa GraphQL:n sijasta. Valitettavasti samanlaista asetusta ei löydy GraphQL-lisäosalle, minkä takia pitää ladata uusi lisäosa, joka tuo samanlaiset asetukset ACF-lisäosaan. Lisäosa mahdollistaa ACF-lisäkenttien sisällyttämisen GraphQL:n välittämiin tietoihin. Kyseinen lisäosa tunnetaan nimellä ”wp-graphql-acf”, joka pitää ladata GitHubista (WPGraphQL for Advanced Custom Fields n.d). Lisäosa on WPGraphQL:n ja Jason Bahlin julkaisema.

Lisäosien kautta lisätyt kentät, CPT:t ja taksonomiat tulevat muuttamaan minkälaisiin tietoihin Next.js-toteutuksella on pääsy ja minkälaiseen lopputulokseen kehitystyössä päädytään. Kuviossa 4 näkyy miltä lopullinen WordPress-näkymä näyttää lisäosien asennuksen jälkeen. Kehitystyössä

käytetään seuraavia CPT:tä; ”Claimed structures” ja ”Natural landscapes”. Näihin CPT:hin on lisätty ACF-lisäosan kautta ”age”- ja ”claimed year”-kentät riippuen kumpi CPT-artikkeli on kyseessä.



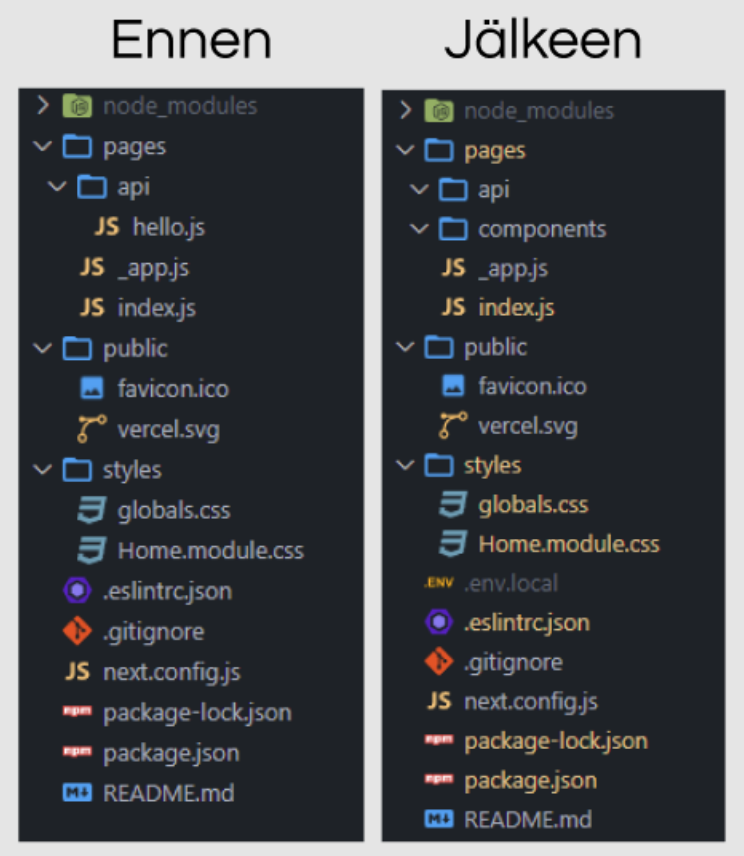
Kuvio 4. Viimeistelty WordPress-ympäristö

Next.js

Next.js-sovelluksen luonti suositellulla menetelmällä lataa tarvittavat JavaScript-paketit ja luo yksinkertaisen kansiorakenteen, joka sisältää suurimmaksi osakseen tiedostoja, jotka sisältävät Next.js-frameworkin resurssilinkkejä ja esimerkkikoodia. Tämä on hyvä lähtökohta henkilöille, jotka ovat juuri aloittaneet Next.js-ympäristöön tutustumisen, mutta opinnäytetyön ja useimpien projektien kannalta Next.js:n tarjoama aloitussovellus sisältää ylimääräistä sisältöä.

Seuraavana vaiheena olisi poistaa tiedostot ja sisältö, jotka eivät ole oleellisia sovelluksessa. Kuvissa 5 on kuvattu miltä lopullinen tiedostorakenne tulee näyttämään verrattuna alkuperäiseen.

”Pages”-kansio tulee tarvitsemaan muutaman muutoksen, kuten ”api”-kansion sisältämä ”hello”-tiedosto tulisi poistaa ja ”Pages”-kansioon tulisi lisätä uusi kansio nimeltä ”components”, joka tulee sisältämään myöhemmin luotavat uudelleen käytettävät sovelluksen osat, joihin saattaa kuulua esim. header-, footer- ja posts-komponentti. ”Public”-kansion sisältämien tiedostojen poistaminen ei ole välttämätöntä, mutta jos ne halutaan poistaa, tulee huomioida, että se tulee luomaan virheitä sovelluksessa, jonka vuoksi on suositeltavaa muuttaa sovelluksen koodia ennen näiden tiedostojen poistamista. Viimeisenä tiedostorakennemuutoksena Next.js-sovelluksen juureen voidaan lisätä ”.env.local”-tiedosto, joka on Next.js-ympäristössä toimiva ympäristömuuttuja-tiedosto. Kyseinen tiedosto mahdollistaa arkaluotoisien arvojen piilottamisen, jotta sovelluksen toimivuuden kannalta kriittiset osat pystyttäisiin pitämään turvassa. Tällaisia arvoja ovat mm. api-avaimet, tietokannan osoite ja tähän kehitystyöhön oleellinen GraphQL-päätepiste.



Kuvio 5. Alkuperäinen ja lopullinen tiedostorakenne

Tiedostorakenteen muuttamisen jälkeen voidaan yksinkertaistaa sovelluksen koodia. Suurin osa Next.js-sovelluksen koodista sijaitsee sivujen templaatti-tiedostoissa, kuten ”index.js”-tiedostoissa

sekä komponenttitiedostoissa. Koska sovelluspohja ei sisällä komponentteja pitää muokata vain ”index.js”-tiedoston koodia. Kuviossa 6 on kuvattu miltä ”index.js”-tiedoston tulisi näyttää, kun ylimääräinen koodi on poistettu. Tiedoston yksinkertaistukseksi riittää, että ”Home”-funktion palauttamasta koodista poistetaan kaikki ulommaisen ”div”-tagin sisältämä koodi, jonka jälkeen nettiselain tulee näyttämään tyhjää sivua, jonka vuoksi on kannattavaa lisätä jotain sisältöä sivulle mm. kuvion 6 sisältämän esimerkin mukaisesti. Tiedostossa käyttämättömien Next.js-komponenttien import-lausekkeiden poistaminen ei ole tarpeellista, koska niitä voidaan hyödyntämään myöhemmin.

Tiedostossa sijaitseva ”Home”-funktio sisältää normaalilta HTML-koodilta vaikuttavaa koodia, joka todellisuudessa on JSX-koodia (kok. JavaScript XML). JSX-koodi mahdollistaa HTML-koodin kirjoittamisen JavaScript-tiedostoon, jonka avulla voidaan myös upottaa arvoja suoraan HTML-tageihin. JSX-koodi ei mahdollista täydellisen HTML-koodin kirjoittamista, jonka vuoksi joidenkin HTML-attribuuttien nimet saattavat poiketa. Kuviossa 6 näkyy mm. miten ”div”-tagin ”class”-attribuutti esiintyy JSX-koodissa ”className” nimellä. Kyseinen eroavaisuus on välttämätöntä, koska React pohjautuu JavaScript-ohjelmointikieleen, jossa on tiettyjä avain sanoja, joita ei tulisi käyttää, joihin kuuluu sana ”class” (Why React uses className over class attribute 2021).

```
import Head from 'next/head'
import Image from 'next/image'
import styles from '../styles/Home.module.css'

export default function Home() {
  return (
    <div className={styles.container}>
      <h1>Everything is in order</h1>
    </div>
  )
}
```

Kuvio 6. Index.js-tiedoston koodiesimerkki

”Pages”-kansiossa sijaitseva ”_app.js”-tiedosto ei tule tarvitsemaan muutoksia tässä suhteellisen yksinkertaisessa kehitystyössä, mutta sen olemassaolo on erittäin tärkeää. Kyseinen tiedosto on uloin templaatti-tiedosto, joka tulee sisältämään kaikki muut templaatti-tiedostot, kuten aiemmin muokatun ”index”-tiedoston. Tiedostoon voidaan lisätä mm. globaaleja tyylitiedostoja, joita käytetään jokaisella sovelluksen sivulla.

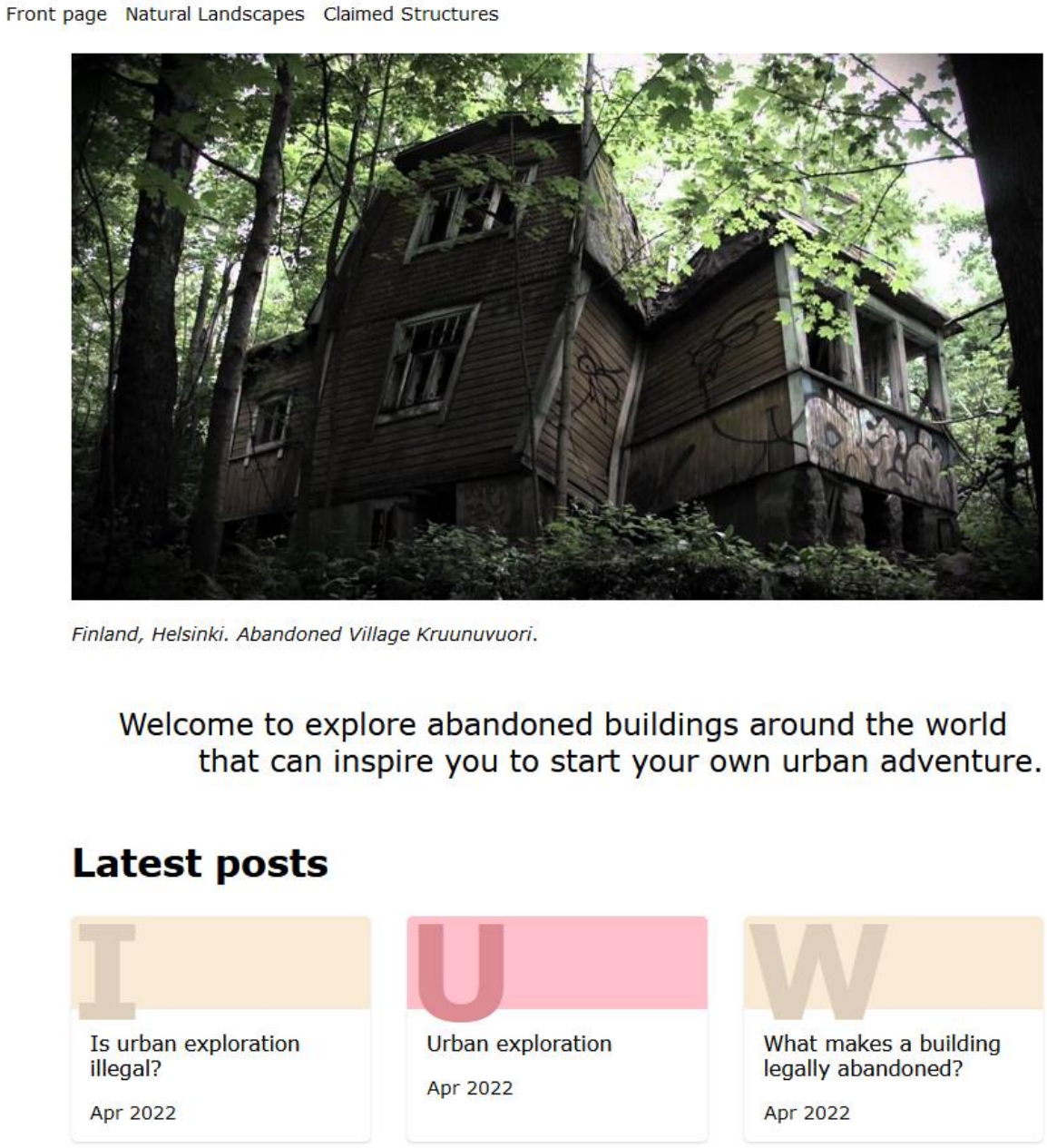
Viimeiseksi sovellukseen voidaan lisätä WordPressin GraphQL-päätepiste ”.env.local”-tiedostoon. Muistutuksena, että päätepisteen osoite löytyy WordPressin GraphQL-lisäosan asetuksista. ”.env.local”-tiedosto on todella yksinkertainen, jonka vuoksi se voi sisältää vain muuttujan nimen ja sen arvon esim. ”WP_ENDPOINT=http://oppari.test/graphql”. Huomiona, että arvo tullaan tallentamaan merkkijonoksi.

6.2 Kehitettävä sovellus

Kehitystyössä kehiteltävä headless-ympäristö tulee sisältämään ympäristön pystytyksen lisäksi suhteellisen yksinkertaisen headless-sovelluspohjan kehitystä, joka on rakenteellisesti ja ominaisuuksien osalta lähes identtinen WordPressin ”Twenty Twenty-One”-teeman kanssa. Tämän tarkoitus on varmistaa, ettei sovellus ole liian yksinkertainen tai liian pitkälle kehitetty, mikä vaikuttaisi minkälaisissa headless-toteutuksissa kyseistä pohjaa voitaisiin käyttää.

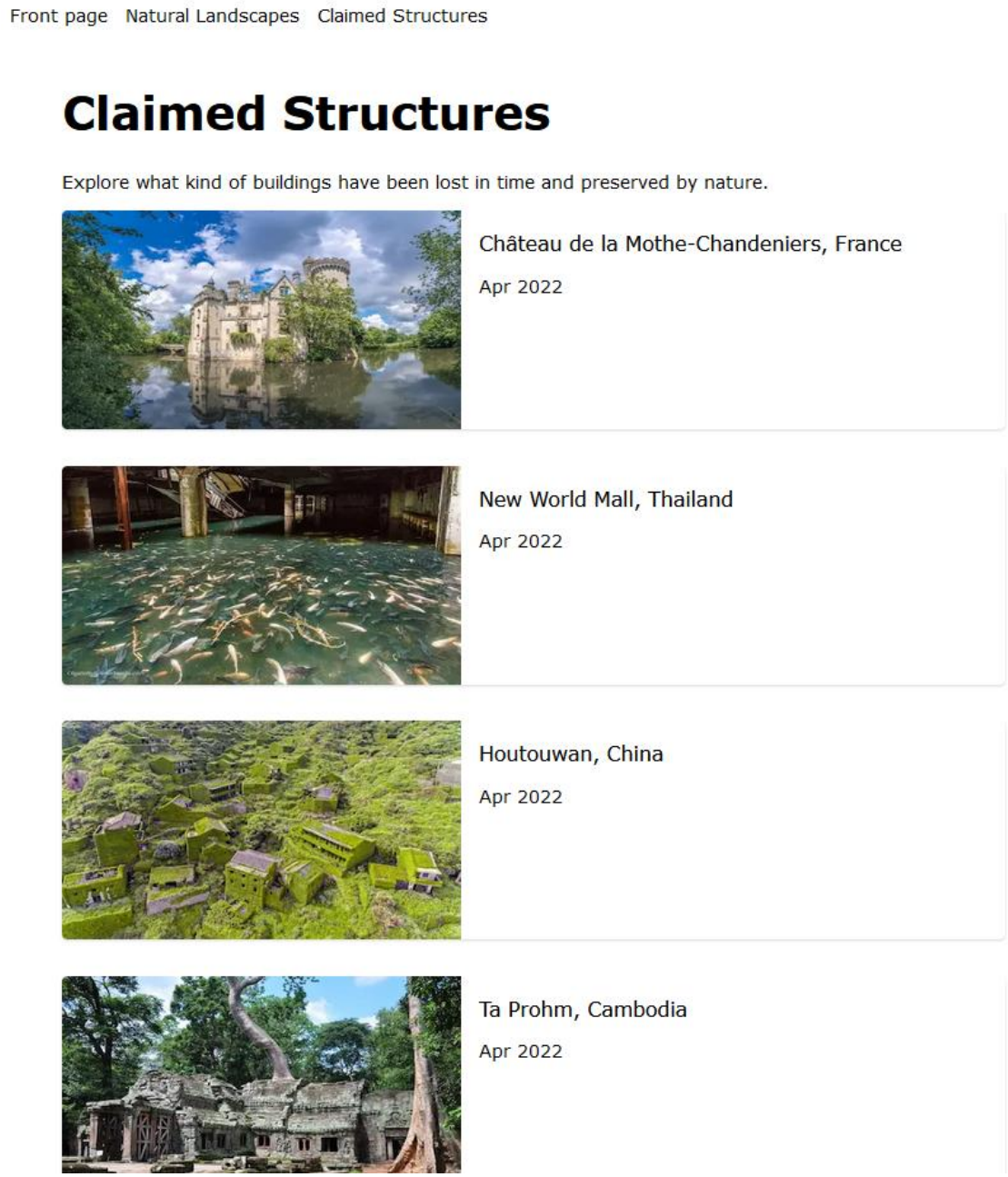
Sovelluspohja sisältää perinteisen sivurakenteen, joka on tuttu monilta sivustoilta, mitä käytetään myös monessa WordPress teemassa, kuten aiemmin mainitussa ”Twenty Twenty-One”-teemassa. Käytännössä sovelluksen sivurakenne tulee sisältämään kolme päätemplaattia, jotka tulevat olemaan Etusivu, arkistosivu ja artikkelisivu, johon tullaan jatkossa viittaamaan ”yksittäinen sivu”-termillä.

Kuviossa 7 on kuvattu miltä sovelluksen etusivu voisi näyttää, kun sovelluspohjan etusivuun on lisätty tarpeelliset CSS-muutokset ja sisältöä on lisätty WordPressin kautta. Etusivu sisältää myös Next.js-sovellukseen rakennettuja komponentteja kuten navigaatiovalikon, joka hakee sisältönsä WordPressistä. Tämän lisäksi etusivu sisältää posts-komponentin, joka näyttää WordPressin sisältämät artikkelit halutussa muodossa.



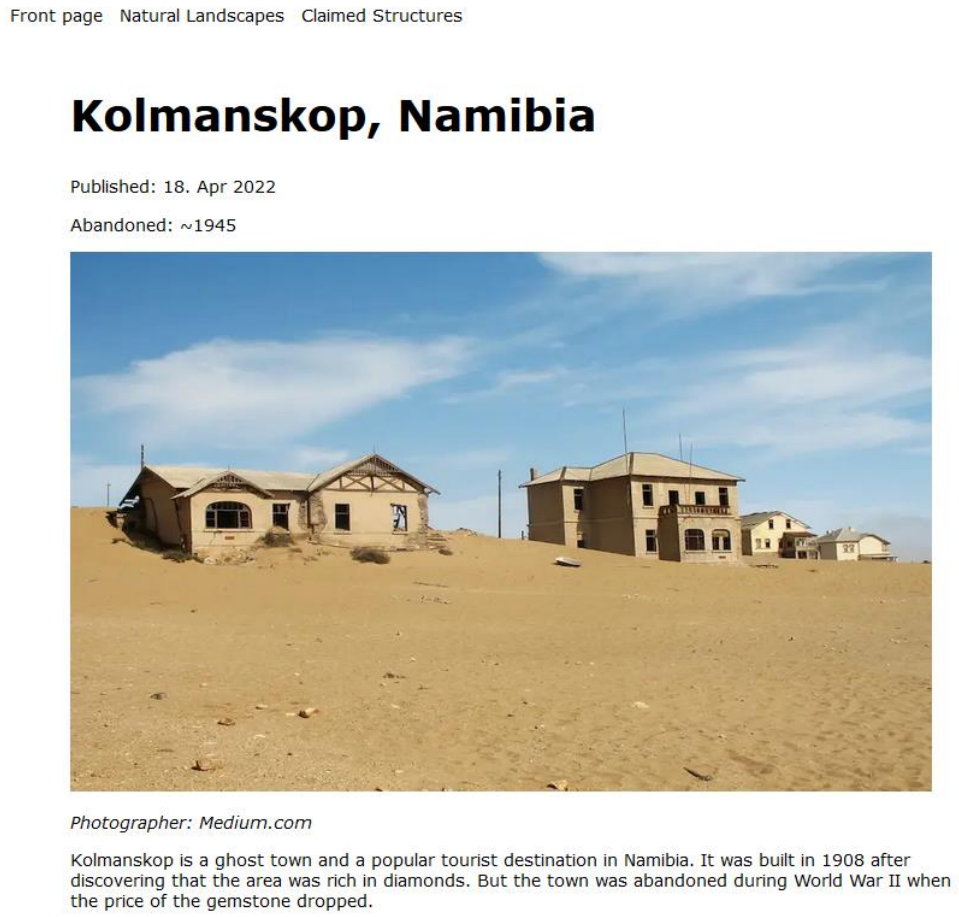
Kuvio 7. Sovelluksen viimeistelty etusivu

Arkistosivu on nimensä mukaisesti sivu, joka sisältää kaikki kyseiseen arkistoon kuuluvat artikkelit. Kuviossa 8 näkyy, että arkistosivu hyödyntää etusivulla käytettyä posts-komponenttia listaamaan artikkeleita halutussa muodossa, minkä tyylitystä on muokattu, jotta komponentti sopisi paremmin artikkelisivulle. Kuviossa 8 näkyy myös, että sivu hakee WordPressistä ”Claimed Structures”-sivun sisällön, joka näytetään posts-komponentin yläpuolella.



Kuvio 8. Sovelluksen viimeistelty arkistosivu

Kuviossa 9 esitellään sovelluksen viimeinen templaatti, jonka avulla pystytään näyttämään artikkeleiden ja sivujen sisältöä. Sovellus sisältää muutaman variaation templaatista artikkelisivun sisältävien ACF-kenttien vuoksi. Kuviossa 9 näkyy, miten artikkelissa esitellään ensin artikkelisivun sisältämät ACF-kentät, jotka sisältävät seuraavat tiedot: vuosi milloin alue on hylätty, kuvan osoite ja kuka on ottanut artikkelin kuvan. Templaatin toista variaatiota käytetään, kun kyseessä on jokin muu sivu, joka ei sisällä ACF-kenttiä.



Kuvio 9. Sovelluksen viimeistelty artikkelisivu

Työn lopputuloksena syntyy luvussa kuvailtu sovelluspohja, jonka varaan voidaan rakentaa sovellus, joka hyödyntää sovelluspohjan tarjoamaa rakennetta ja ominaisuuksia. Sovelluspohjan tarkoituksena on toimia sovelluspohjana sovelluksissa, jotka tarvitsevat samanlaisen sivurakenteen ja samanlaisia ominaisuuksia, minkä myötä varsinainen sovelluskehitys voitaisiin aloittaa mahdollisimman nopeasti.

6.3 Headless-sovelluksen kehittäminen

Kehitystyön ympäristön backendin ja frontendin pystytyksen jälkeen voidaan yhdistää kyseiset osat toisiinsa, jotta saadaan headless-toteutuksen nimen mukainen ympäristö. Mikäli WordPressin asetukset on määritelty oikein, backend ei tule tarvitsemaan muita muutoksia. Kappaleen sisältö tulee koskemaan pääsääntöisesti Next.js-sovellusta.

GraphQL-kyselyt

Frontendin yhdistäminen WordPress-backendiin voidaan todeta onnistuneeksi, kun frontend kykenee tekemään kyselyitä backendiin, jonka perusteella backend välittää tarvittavat tiedot frontendille. Aiemman luvun lopussa lisättiin GraphQL:n päätepiste Next.js-sovellukseen, joten seuraavaksi pitäisi luoda api, joka tekee kyselyitä kyseiseen päätepisteeseen. Next.js-sovelluksen ”pages”-kansio sisältää ”api”-kansion, jonka sisään voidaan luoda uusi JavaScript-tiedosto, jonka tehtävänä on käsitellä GraphQL-kyselyitä.

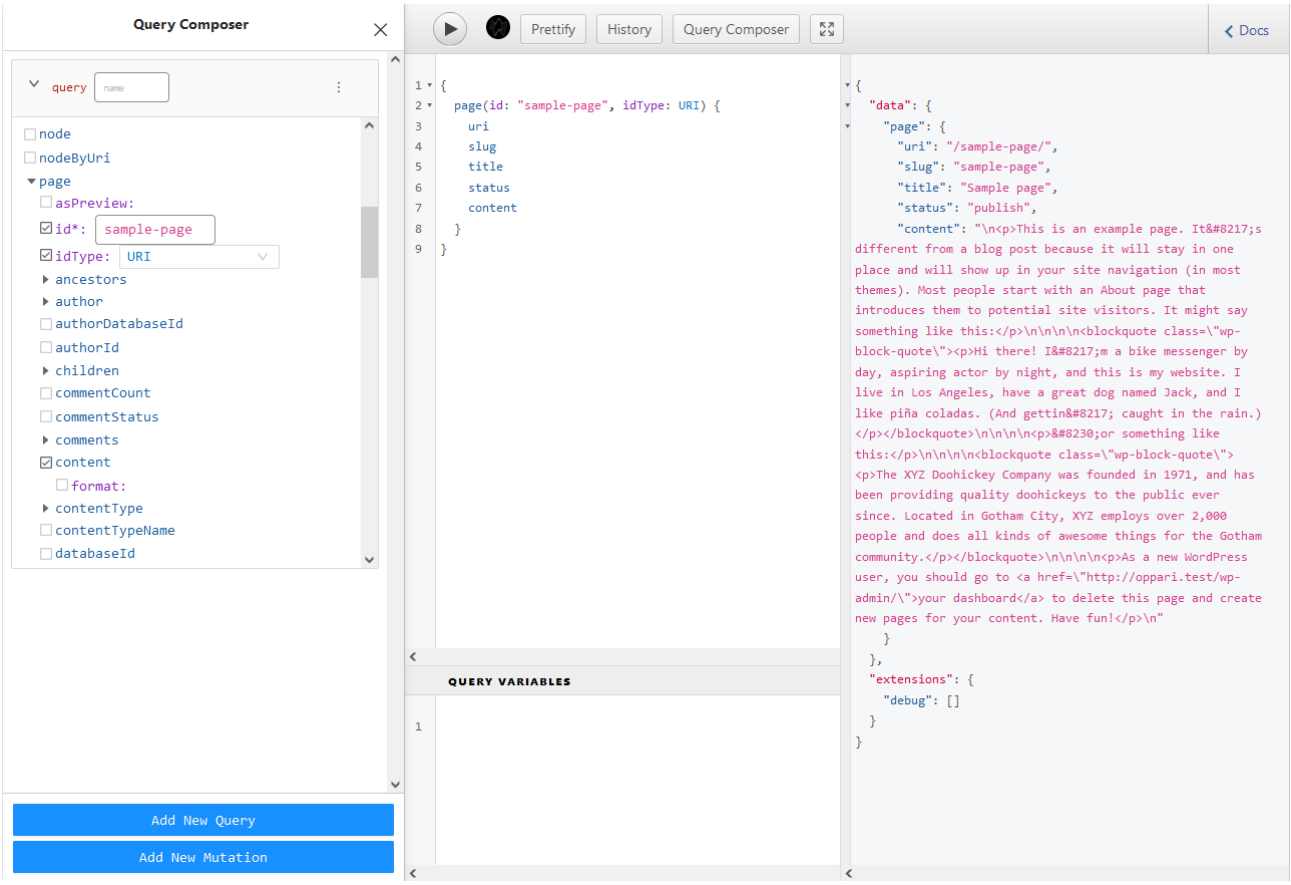
GraphQL-api tulee tarvitsemaan päätepisteen url-osoitteen, joka voidaan hakea seuraavasti ”process.env.muuttujan_nimi”. Tämän jälkeen voidaan luoda funktio, joka tekee kyselyn GraphQL:n päätepisteeseen. Kuviossa 10 on esitelty Kendalin (2020) käyttämä ”fetchAPI”-funktio, joka tarvitsee kyselyparametrin suorittaakseen kyselyn onnistuneesti. Kyseinen kysely lähetetään JavaScriptin sisältämän fetch-funktion avulla WordPressin GraphQL-lisäosaan prosessoitavaksi, mihin GraphQL vastaa palauttamalla halutut tiedot apille, mitkä api välittää templaatille. Mikäli ”fetchAPI”-funktio havaitsee virheviestin GraphQL:n antamissa tiedoissa api tulee tekemään virheviestin. GraphQL-api:n virheviestin vastausta voisi muuttaa palauttamalla virhetilanteissa GraphQL:n tarjoaman virheviestin sen sijasta, että luodaan virheviesti, joka estää sovelluksen toiminnan.

```
const API_URL = process.env.WP_ENDPOINT

const fetchAPI = async (query, { variables } = {}) => {
  const headers = { 'Content-Type': 'application/json' }

  const response = await fetch(API_URL, {
    method: 'POST',
    headers,
    body: JSON.stringify({
      query,
      variables
    })
  })

  const json = await response.json()

  if (json.errors) {
    console.log(json.errors)
    console.log(query, variables)
    throw new Error('Failed to fetch')
  }

  return json.data
}
```

Kuvio 10. “Graphql.js”-tiedoston fetch-funktio

Pelkkä ”fetchAPI”-funktio riittää kutsun suorittamiseen, mutta jotta koodi olisi mahdollisimman selvää, helposti ylläpidettävää ja kaikki kutsuihin liittyvä koodi löytyisi luodusta api-tiedostosta on tärkeää lisätä funktiot samaan tiedostoon ”fetchAPI”-funktion kanssa.

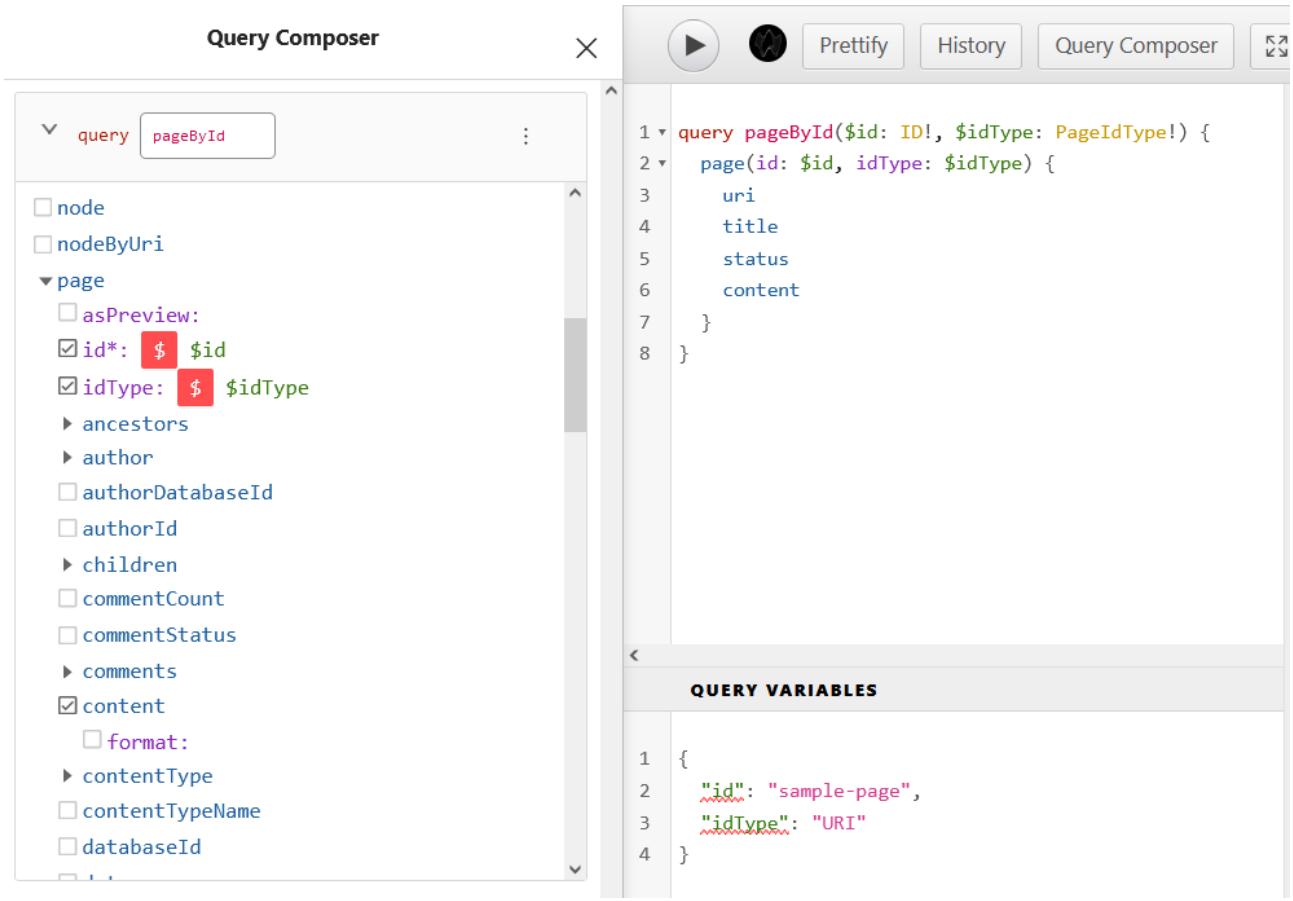
Kyselyiden tekemisessä ja testaamisessa kannattaa hyödyntää WordPressin GraphQL-lisäosan sisältämää ”GraphiQL IDE”-editoria, jonka avulla voidaan tehdä testikyselyitä WordPressin tietokantaan. ”Query Composer”-painike avaa uuden valikon, jonka avulla voidaan lisätä mitä sisältöä halutaan tuoda tietokannasta. Kuvio 11 sisältää esimerkin kyselystä, joka hakee WordPressin ”sample-page”-sivun uri-osoitteen, slugin, otsikon, tilan ja sisällön.



Kuvio 11. WordPress GraphQL-lisäosan "GraphiQL IDE"-näkymä

Kun kyselyyn halutaan lisätä muuttujia, ne voidaan lisätä ”Query Variables”-osion kautta. Muuttujien lisääminen tulee myös muuttamaan kyselyn syntaksia. Kuviossa 12 on luotu query-funktio, joka sisältää ”id”-parametrin, joka määrittää mitä sivua kysely hakee. Parametrin ”idType”:n tehtävänä on määrittää mihin arvoon id:n arvoa verrataan. Esimerkissä id:n arvoksi on määritetty

”sample-page”, jota verrataan uri-osoitteeseen, joten palautettava arvo pitäisi olla sama kuin kuvi-ossa 11.



Kuvio 12. Muuttujien käyttäminen GraphQL-kyselyssä

Kehitystyössä ei tulla käsittelemään GraphQL-kyselykieltä tämän tarkemmin, jonka vuoksi kieleen kannattaa perehtyä tarkemmin WPGraphQL- ja GraphQL-sivujen kautta.

Kun toiminnallinen kysely on tehty ja todettu toimivaksi se voidaan ongelmitta syöttää Next.js-sovelluksessa sijaitsevaan GraphQL-apiin. Kuviossa 13 on luotu uusi funktio ”getPage”, jolla on URI-parametri, jonka kautta määritellään mitä sivua kysely hakee. Kysely hyödyntää aiemmin tehtyä ”fetchAPI”-funktiota, joka vaatii kyselyargumentin ja tarvittaessa olion toiseksi argumentiksi, joka sisältää kyselyn vaatimat muuttujat. Huomiona, ettei itse ”fetchApi”-funktiota viedä GraphQL-apin ulkopuolelle.

```
// Käytetään sivujen hakemisessa
export const getPage = async (URI) => {
  const data = await fetchAPI(`
    query pageByURI($id: ID!, $idType: PageIdType!) {
      page(id: $id, idType: $idType) {
        uri
        title
        status
        content
      }
    }
    `,
    {
      variables: {
        id: URI,
        idType: 'URI'
      }
    }
  )
  return data
}
```

Kuvio 13. ”Graphql.js”-tiedoston ”getPage”-funktio

Kyselyfunktioiden määrityksen jälkeen voidaan keskittyä dynaamisen sisällön tuontiin Next.js-sovellukseen, jonka myötä todetaan myös onko päätepiste määritetty oikein ja sisältääkö ”.env.local”-tiedosto oikean linkin. Viemällä ”Index.js”-tiedostoon aiemmin luotu ”getPage”-funktio tulee antamaan lopullisen varmistuksen onko GraphQL-api tehty oikein. Kuviossa 14 näkyy miten ”getPage”-funktiota käytetään ”getStaticProps”-funktion sisällä, mikä on Next.js:n sisään rakennettu funktio, jota käyttämällä voidaan hakea ja määritellä uusia arvoja, jotka välitetään ”props”-olion sisällä templaattiin sivun prosessoinnin yhteydessä (GetStaticProps n.d).

Kun sivu on onnistuneesti prosessoitu ja sovellus näyttää onnistuneesti dynaamista sisältöä WordPressistä, voidaan todeta headless-ympäristö toimivaksi. Next.js-sovellukseen välitetty sisältö saattaa sisältää HTML-tagejä, joka tekee sisällöstä vaikealukuista. Kyseinen sisältö on todennäköisesti WordPressin Gutenberg-editorin avulla luotu, jonka takia kyseinen sisältö pitää käsitellä, jotta HTML-tagit tunnistetaan onnistuneesti GraphQL:n tarjoamasta merkkijonosta. React

sisältää hyödyllisen attribuutin tällaisia tilanteita varten, jonka nimi on ”dangerouslySetInnerHTML”, joka nimensä mukaisesti tulee upottamaan HTML-tagin sisälle kaikki merkkijonon sisältämät HTML-tagit. Lopputuloksena pitäisi olla hyvin asettuva ja helppolukuinen sisältö.

```
import Head from 'next/head'
import Image from 'next/image'
import styles from '../styles/Home.module.css'

import { getPage } from './api/graphql'

export default function Home({ frontpage }) {

  return (
    <div className={styles.container}>
      { frontpage.page.content }

      <div dangerouslySetInnerHTML={{ __html: frontpage.page.content }} />
    </div>
  )
}

export const getStaticProps = async () => {
  let frontpage = await getPage('sample-page')

  return {
    props:{
      frontpage
    }
  }
}
```

Kuvio 14. "Index.js"-tiedosto WordPress-tietojen upottamisen jälkeen.

Kun WordPressistä välitetty tieto välittyy oikein Next.js-sovellukseen voidaan siirtyä varsinaiseen sovelluskehitykseen ilman ongelmia siitä tulevatko GraphQL-kutsut toimimaan tai tuleeko tietojen upotuksessa ongelmia, jotka voisivat vaikuttaa sovelluskehityksen edistymistä

Sovelluksen rakenne

Sovelluksen tarkoituksena on tuoda WordPressin teemasta mahdollisimman monta ominaisuutta, jotta Next.js-sovellus olisi mahdollisimman yhtenäinen WordPress-teeman kanssa. WordPressin tarjoamat teemat ovat yksinkertaisia, mikä tekee tästä kehitystyöstä suhteellisen yksinkertaista. Sovelluksen pitäisi sisältää seuraavan sivurakenteen; etusivu > arkisto > yksittäinen sivu. Sivurakenteen lisäksi sovelluksen pitäisi myös hakea kaikki julkaistut sivut, jonka myötä suurin osa sovelluksen sivuista tulisivat olemaan dynaamisia, mitkä luodaan sen perusteella, kuinka monen sivun tietoja GraphQL välittää sovellukselle ja ainoana staattisena sivuna tulisi olemaan etusivu, joka saa kuitenkin sisältönsä dynaamisesti WordPressin ”frontpage”-sivulta.

Etusivu on yleensä kaikista tärkein sivu ensivaikutelman luomisen suhteen, minkä vuoksi on hyödyllistä tehdä siitä uniikki, joka tekee muutoksien tekemisestä mahdollisimman helppoa verrattuna muihin sivuihin, jotka tulevat sisältämään samanlaisessa muodossa tietoa. Jotta sovellus olisi mahdollisimman helppo päivittää WordPressillä, se tulee sisältämään dynaamisia arkistosivuja, joita luodaan staattisesti Next.js-sovelluksessa. Vaikka arkistosivuja voidaan lisätä myös dynaamisesti se ei ole oleellinen ominaisuus, jotta sovellus olisi käyttökelvollinen. WordPressin sisältämiä sivuja voidaan kategorisoida eri tyyppeihin määrittämällä ACF-lisäosan kautta arvo, jonka avulla voidaan määritellä sivun tyyppi. Sovelluksessa luodaan myös staattinen arkisto, joka sisältää kaikki kategorioimattomat artikkelit. Kommentoiminen ei ole myöskään oleellinen ominaisuus, minkä vuoksi kehitystyössä ei tulla kehittämään kommentointi ominaisuutta. WordPress sisältää myös ominaisuuden, jonka kautta voidaan luoda uusia ja muokata olemassa olevia valikkoja, minkä vuoksi Next.js-sovelluksen pitäisi hyödyntää kyseistä ominaisuutta.

Sovelluksen sivurakenteen kehittäminen

Next.js-framework on kehitelty olemaan kehittäjäystävällinen ympäristö, jonka voi nähdä mm. miten sivuston rakenne määritellään. Next.js ottaa sovelluksen prosessoinnin aikana kaikki ”pages”-kansion sisältämät tiedostot sekä kansiot ja luo näiden pohjalta sivustorakenteen, jossa ”index”-tiedosto tulee olemaan polun oletustiedosto. Muun muassa ”pages”-kansion juuressa sijaitseva ”index”-tiedosto omaa ”localhost:3000/”-osoitteen, joka on sovelluksen etusivu ja myöhemmin

lisättävä ”posts”-kansio tulee sisältämään oman ”index”-tiedoston, joka omaa ”localhost:3000/posts”-osoitteen. Kansioiden kautta voidaan myös luoda alasivuja esim. ”posts”-kansiossa oleva tiedosto voidaan nimetä muulla nimellä kuin ”index” esimerkiksi tiedosto voidaan nimetä ”hello-world.js”-tiedostoksi, jonka sisältöä voidaan katsella ”localhost:3000/posts/hello-world”-osoitteessa.

Laajennetaan tämänhetkistä sivurakennetta lisäämällä edellisessä kappaleessa mainitut kansiot ja tiedostot. ”Pages”-kansion sisään tullaan lisäämään uusi kansio ”posts”, joka tulee sisältämään kaksi tiedostoa; dynaamisen ”[slug].js”-tiedoston ja staattisen ”index.js”-tiedoston. ”Index.js”-tiedosto tulee toimimaan arkistosivuna, joka tulee sisältämään artikkeleita, jotka eivät ole missään kategoriassa ja ”[slug].js”-tiedosto tulee generoimaan yksittäisiä sivuja kyseisille artikkeleille.

Kuvio 15 sisältää ”posts”-kansion ”index.js”-tiedoston koodin, joka sisältää uuden komponentin ”Posts” ja uuden GraphQL-api funktion ”getAllPosts”. ”Posts”-komponentin tehtävänä on käydä läpi ”posts”-muuttujan sisältämät oliot ja lisätä ne sivulle allekkain listaksi. ”GetAllPosts”-funktio tulee noutamaan GraphQL:stä kaikki kategoriattomat artikkelit.

```
import Head from 'next/head'
import Link from 'next/link'

import Posts from '../../src/components/posts'

import { getAllPosts } from '../api/graphql'

// URL: domain/posts
export default function Blog({ posts }) {
  return(
    <div>
      <main>
        <h2>All posts</h2>

        <Posts posts={posts} />
      </main>
    </div>
  )
}

export async function getStaticProps() {
  const posts = await getAllPosts()
  return {
    props: {
      posts
    }
  }
}
```

Kuvio 15. "Posts"-kansion "index.js"-tiedoston koodi

Kuviossa 16 näkyy miten ”getAllPosts”-funktio on muodostettu. Funktio ei sisällä parametreja, joten kyselyn tulokseen ei voida vaikuttaa ”index.js”-tiedostosta. Itse kysely on yksinkertainen, johon on lisätty ehtoja, kuinka monta artikkelia haetaan kerralla ja millaisessa järjestyksessä ne tulisi palauttaa.

```
export const getAllPosts = async () => {
  const data = await fetchAPI(`
    query AllPosts {
      posts(first: 20, where: { orderby: { field: DATE, order: DESC}}) {
        edges {
          node {
            id
            date
            slug
            title
            status
          }
        }
      }
    }
  `)
  return data?.posts?.edges
}
```

Kuvio 16. GraphQL-apin uusi funktio "getAllPosts"

Jotta koodi säilyisi selkeänä ja tiedostorakenne olisi järkevä, Next.js-projektin juureen tehdään uusi kansio nimeltä ”src”. Kansio tulee sisältämään muun muassa komponentteja, joten lisätään ”src”-kansion sisään aiemmin luotu ”components”-kansio. Komponenttien uudelleensijoittaminen ”pages”-kansion ulkopuolelle estää Next.js:ää luomasta sivuja komponenteille. Periaatteessa ”api”-kansiokin voitaisiin siirtää ”src”-kansioon, mutta kasio ei sisällä templaatti-tiedostoja, jotka loisivat visuaalisesti näkyvää sivua.

Kuvio 17 sisältää ”Posts”-komponentin koodin, joka luo HTML-elementtejä artikkeleiden lukumäärän mukaan. Kuviossa on myös nostettu esille, että JSX-koodissa esiintyy tilanteita, joissa ei kuulu käyttää normaalia JavaScriptin syntaksia. Tämän vuoksi map-metodin funktion aaltosulku on korvattu normaalilla sululla.

```
import Link from "next/link"

export default function Posts({ posts }) {
  // console.log(posts[0])

  return (
    <>
      {
      // Huom. map-metodin funktio tulee sisällyttää "(" eikä "{"
      (posts) ?
        posts.map( (post) => (
          (post.node.status == 'publish') ?
          <div key={ post.node.id }>
            <Link href={`/posts/${ post.node.slug }`}>
            <h3>
              <a>{ post.node.title }</a>
            </h3>
            </Link>
            <p>{ post.node.author }</p>
          </div>
        :
          ''
        ))
      :
        <p>No posts found</p>
      }
    </>
  )
}
```

Kuvio 17. "Posts"-komponentin koodi

”Index.js”-tiedoston jälkeen voidaan siirtyä ”[slug].js”-tiedostoon. Huomio ettei kaikki dynaamiset tiedostot tule olemaan ”slug” nimisiä vaan ne tulisi nimetä sen mukaan minkä arvon mukaan niitä generoidaan. Tässä tapauksessa tiedosto tulee nimeämään itsensä ”slug”-arvon mukaan, mikä tulee olemaan jokaisessa artikkelissa eri. ”Slug”-arvon sijasta voitaisiin myös käyttää ID-arvoa, joka puolestaan vaikuttaisi myös kuinka luettava lopullinen url-osoite tulee olemaan.

”Slug”-tiedosto sisältää uutena funktiona ”getStaticPaths” ja uuden olion ”useRouter”, kuten kuviossa 18 näytetään. ”GetStaticPaths”-funktion avulla alustetaan ”slug”-tiedostosta pohjautuvat uudet sivut. Jokainen artikkeli tulee saamaan oman url-osoitteen ”slug”-arvon mukaan. Kuviossa 18 rivillä 12 tarkastetaan, onko ”isFallback”-arvo epätosi, joka tarkoittaisi, että Next.js on viimeistellyt sovelluksen prosessoinnin ja kaikki sivut on luotu. Mikäli arvo on epätosi eikä artikkelin ”slug”-arvoa löydy sovellus tulee palauttamaan 404-virheviestin, joka tarkoittaa, ettei sivua löytynyt (GetS-

taticPaths n.d). Sen sijasta, että if-lauseke palauttaisi 404-virheviestin se palauttaa lauseen ”Something went wrong”. Oikeaoppisesti 404-virhetilanteissa tulisi palauttaa ”404.js”-templaatin, joka korvaisi ”Something went wrong”-lauseen. Viimeisenä huomiona templaatin sisältämä if-lauseke varmistaa, ettei sivun sisältöä näytetä, mikäli sivuston generointi on kesken.

```
import Head from 'next/head'
import Link from 'next/link'

import { useRouter } from 'next/router'

import { getAllPostsWithSlug, getPost } from '../api/graphql'

// URL: localhost:3000/posts/[PostSlug]
export default function Post({ post }) {
  const router = useRouter()

  if(!router.isFallback && !post?.slug) {
    return <p>Something went wrong</p>
  }

  return (
    <div>
      <main>
      {
        (router.isFallback) ?
          <h2>Loading</h2>
        :
          <article>
            <div>
              <h1>{ post.title }</h1>
              <p>{ post.date }</p>
            </div>

            <div dangerouslySetInnerHTML={{ __html: post.content }} />
          </article>
      }
      </main>
    </div>
  )
}

export const getStaticPaths = async () => {
  const posts = await getAllPostsWithSlug()

  return {
    paths: posts.edges.map(({ node }) => `/posts/${ node.slug }`) || [],
    fallback: false
  }
}

export const getStaticProps = async ({ params }) => {
  const data = await getPost(params.slug)

  return {
    props: {
      post: data.post
    }
  }
}
```

Kuvio 18. "[slug.js]"-tiedoston koodi esimerkki

”[slug].js”-tiedoston polut tullaan määrittelemään sen perusteella, kuinka moni sivu tulee sisältämään ”getAllPostsWithSlug”-funktion pyynnön kriteerit. Kuviossa 19 käy ilmi, että kyselyssä haetaan ensimmäiset 1000 artikkelia, joista halutaan vain slug-arvo. Tämä arvo vastaa polkutunnuksen arvoa.

```
export const getAllPostsWithSlug = async () => {
  const data = await fetchAPI(`
    {
      posts(first: 1000) {
        edges {
          node {
            slug
          }
        }
      }
    }
  `)
  return data?.posts
}
```

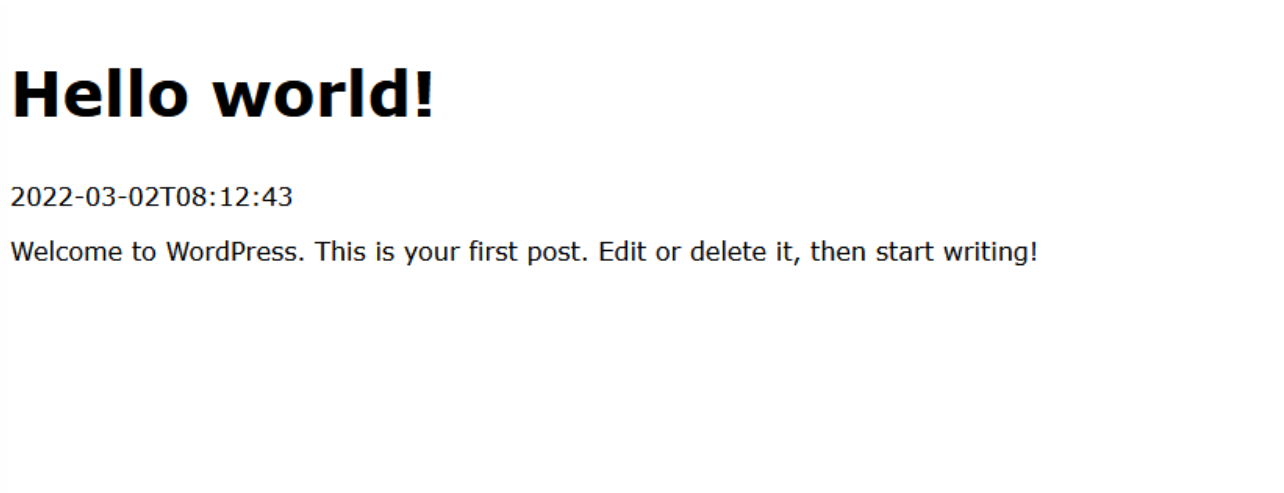
Kuvio 19. GraphQL-apifunktio getAllPostsWithSlug

Kuviossa 18 käy myös ilmi, että GraphQL-apiin on luotu uusi funktio nimeltä ”getPost”, joka tekee kyselyn, johon GraphQL vastaa palauttamalla yhden artikkelin sisällön. Artikkelin sisältö saadaan välittämällä ”slug”-arvo ”getPost”-funktiolle, jonka ”getStaticProps”-funktio saa ”slug”-tiedoston nimestä, mikä määritellään sovelluksen prosessoinnin yhteydessä dynaamisesti kuvion 19:sta esittämällä funktiolla (GetStaticProps n.d). Kuviossa 20 näkyy, ettei ”getPost”-funktion sisältämä kysely poikkea paljoa kyselystä, joka esiintyy kuviossa 13 (s. 38). Kyselyssä on tehty kaksi muokkausta; artikkeleissa verrataan ”slug”-arvoa ”URI”-arvon sijasta ja sivun sijasta etsitään artikkelia.

```
export const getPost = async (slug) => {
  const data = await fetchAPI(`
    query PostBySlug($id: ID!, $idType: PostIdType!) {
      post(id: $id, idType: $idType) {
        title
        excerpt
        slug
        date
        content
      }
    }
    `,
    {
      variables: {
        id: slug,
        idType: 'SLUG'
      }
    }
  )
  return data
}
```

Kuvio 20. "GetPost"-funktion koodi

Tuloksena syntyy kuvion 21 mukainen sivu, joka toimii ”localhost:3000/posts/slug-arvo”-osoitteen alla ja näyttää artikkelin sisältämän sisällön.



Kuvio 21. "Hello world!"-artikkelisivu

Sivujen luonti dynaamisesti

Dynaamisten artikkelisivujen jälkeen on hyvä siirtyä dynaamisien sivujen määritykseen. Kuten artikkelisivuissa tarkoituksena on hakea WordPressistä kategorioimattomat sivut, jotka löytyvät WordPressin admin-näkymän ”sivut”-osiosta. Oletuksena WordPress luo sivun luonnin yhteydessä kaksi sivua; ”Privacy Policy” ja ”Sample Page”. Näiden lisäksi tässä vaiheessa on hyvä määrittää lopullinen etusivu, jota ei luoda dynaamisesti, jotta etusivu olisi helposti muokattavissa ”pages”-kansion juuri ”index”-tiedoston kautta. Polku mitä tässä kehitystyössä ei tulla luomaan on ”front-page”.

”Pages”-kansion juureen tulisi luoda uusi dynaaminen tiedosto nimeltä ”[page].js”. Kuviossa 22 kuvataan ”[page].js”-tiedoston sisältämää koodia, joka on lähes samanlainen kuin ”[slug.js]”-tiedoston koodi. Suurimmat eroavaisuudet ovat muuttujissa ja ”getStaticPaths”-funktion luomassa lopullisessa uri-osoitteessa. Riville 43 on lisätty for-luuppi, jonka avulla filtteröidään sivut, joita ei haluta luoda, joihin kuuluu arkistosivut ja etusivu, jotka luodaan omilla tiedostoilla.

```jsx
import Head from 'next/head'
import Link from 'next/link'

import { useRouter } from 'next/router'

import { getAllPagesWithURI, getPage } from './api/graphql'

// URL: localhost:3000/[PageURI]
export default function Page({ page }) {
  const router = useRouter()

  if(!router.isFallback && !page?.uri) {
    return <p>Something went wrong.</p>
  }

  return (
    <div>
      <main>
      {
        (router.isFallback) ?
          <h2>Loading</h2>
        :
          <article>
            <div>
              <h1>{ page.title }</h1>
              <p>{ page.date }</p>
            </div>

            <div dangerouslySetInnerHTML={{ __html: page.content }} />
          </article>
      }
      </main>
    </div>
  )
}

export const getStaticPaths = async () => {
  const pages = await getAllPagesWithURI()

  const uris = []

  // Exclude unwanted uris
  for(let i in pages.edges) {
    if(pages.edges[i].node.uri !== '/front-page/' &&
    pages.edges[i].node.pageType.pageType !== 'Arkisto') {
      uris.push(`${ pages.edges[i].node.uri }`)
    }
  }

  return {
    paths: uris || [],
    fallback: false
  }
}

export const getStaticProps = async ({ params }) => {
  const data = await getPage(params.page)

  return {
    props: {
      page: data.page
    }
  }
}
```

Kuvio 22. Tiedoston "[page].js" sisältämä koodi

”[page].js” tiedostossa määritellään uudet uri-osoitteet GraphQL-apin ”getAllPagesWithURI”-funktion avulla, mikä hakee ensimmäiset 100 sivua. Sivujen luonnissa ei tulla tarvitsemaan muuta tietoa kuin mikä tulee olemaan niiden osoite. Tämän vuoksi kuviossa 23 esitelty ”getAllPagesWithURI”-funktio hakee vain sivujen uri-osoitteet.

```
export const getAllPagesWithURI = async () => {
  const data = await fetchAPI(`
    {
      pages(first: 100) {
        edges {
          node {
            uri
          }
        }
      }
    }
    `
  )
  return data?.pages
}
```

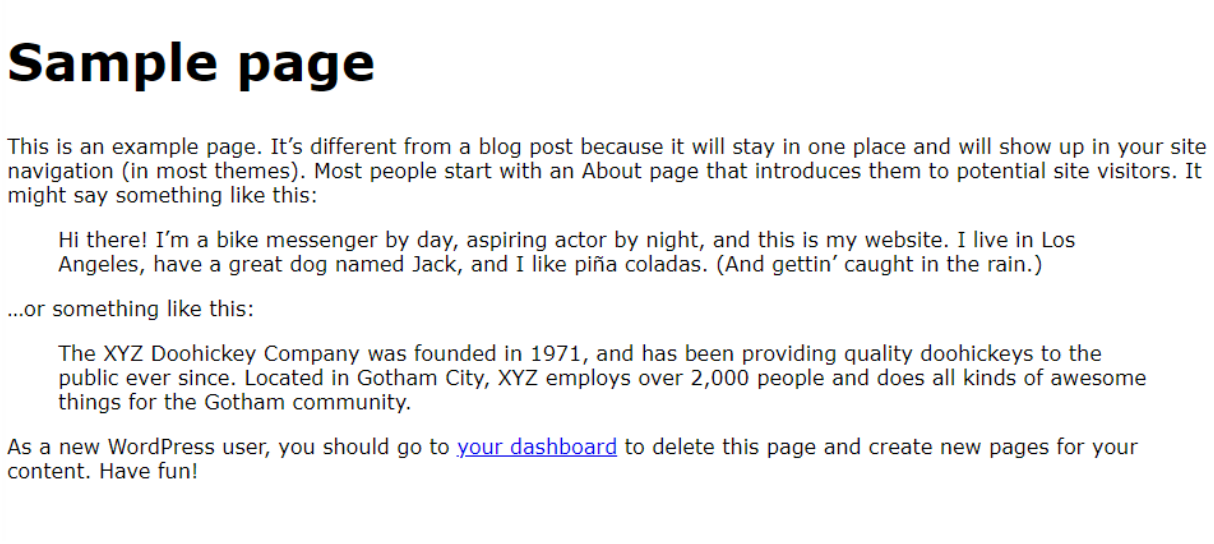
Kuvio 23. GraphQL-apin ”getAllPagesWithURI”-funktio

Kuviossa 22 näkyy myös, että ”getStaticProps”-funktio käyttää ”getPage”-funktiota yksittäisen sivun tietojen hakemisessa, mikä tarvitsee yhden argumentin. Argumenttina käytetään sivun parametria ”page”, minkä arvo vastaa sivun ”uri”-arvoa, joka noudettiin ”getAllPagesWithURI”-funktiolla. Kuvio 24 sisältää ”getPage”-funktion kyselyn, joka sisältää kaksi parametria; id ja idType.

```
export const getPage = async (URI) => {
  const data = await fetchAPI(`
    query pageByURI($id: ID!, $idType: PageIdType!) {
      page(id: $id, idType: $idType) {
        uri
        title
        status
        content
      }
    }
    `,
    {
      variables: {
        id: URI,
        idType: 'URI'
      }
    }
  )
  return data
}
```

Kuvio 24. GraphQL-apin “getPage”-funktio

Lopputuloksena syntyy yksinkertainen sivutemplaatti, joka kykenee näyttämään WordPressistä saatujen sivujen sisällön. Sisältöä voidaan tarkastella ”localhost:3000/slug-arvo”-sivulta. Kuvion 25 esimerkissä näkyy miten ”Sample page”-sivun sisältö näytetään onnistuneesti osoitteessa ”localhost:3000/sample-page”.



Kuvio 25. "[page].js"-tiedostosta luotu yksittäinen sivu

Tällä hetkellä Next.js-sovellukseen on luotu etusivu, kategorioimattomien artikkeleiden arkistosivu, artikkeleiden templaattitiedosto, jota käytetään pohjana, kun luodaan artikkelisivuja, joita luodaan maksimissaan tuhat ja viimeisimpänä lisätty yksittäisien sivujen templaattitiedosto, josta luodaan maksimissaan sata sivua. Sovellus sisältää tällä hetkellä kriittisimmät ominaisuudet, joiden avulla sovelluksessa voidaan vierailla sivuilla, jotka WordPress sisältää ennen muutoksien tekoa. Tästä on hyvä jatkaa lisäämällä sovellukseen käyttökokemusta parantavia ominaisuuksia, joihin kuuluu CPT-arkistot.

6.4 Käyttökokemusta parantavat ominaisuudet

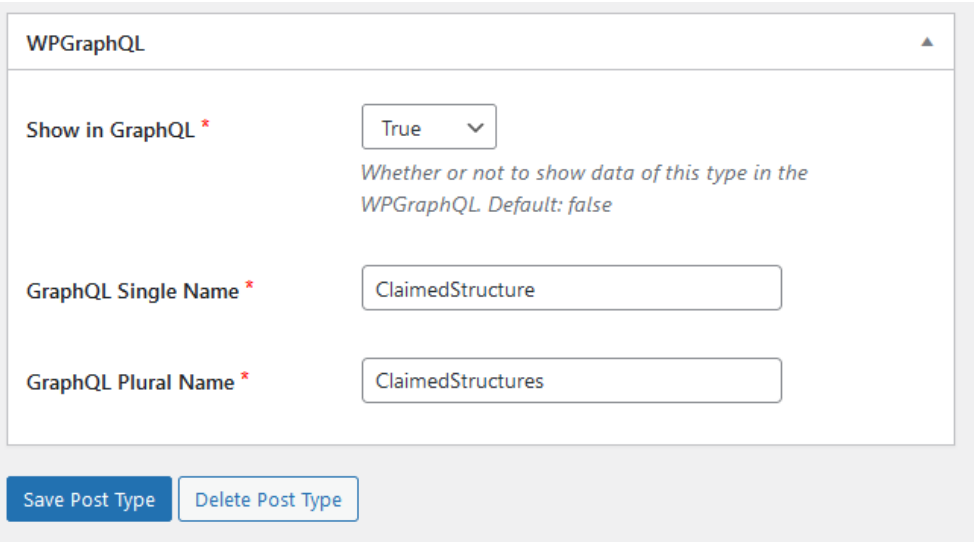
Jotta sovellus olisi, helpommin navigoitavissa sovellukseen tulee lisätä muutama lisäominaisuus. Ensimmäisenä sovellukseen voidaan lisätä CPT-tuki, jonka avulla artikkelien arkistosivuihin voidaan tuoda dynaamisesti sisältöä, kun taas arkisto sivuja luodaan sen perusteella, kuinka moni sivu on määritetty arkistosivuksi ACF-lisäosan luodun kentän avulla WordPressissä. Toisena sovellukseen

voitaisiin lisätä tuki WordPressin navigaatiovalikon noutamiseen, jonka avulla navigaatiota voidaan muokata WordPressin navigaatiovalikon kautta.

CPT-arkistosivu

Vaikka sovellus pystyy noutamaan kriittisimmät sisällöt se ei ole vielä kykeneväinen noutamaan kaikkea WordPressin sisältämää sisältöä. WordPressissä voidaan arkistoida artikkeleita erilaisien CPT:den alle, mitä moni WordPress-sivu hyödyntää, kun halutaan arkistoida erilainen sisältö erilaisen arkiston alle. Tämä tekee CPT-arkistosivusta oleellisen käyttökokemusta parantavan ominaisuuden.

Valitettavasti WPGraphQL:n dokumentaation (Custom Taxonomies n.d) mukaan GraphQL ei kykene noutamaan mitään arvoja CPT:stä ennen kuin kyseiset tyypit on määritetty GraphQL-lisäosalle, joko teeman tai lisäosan kautta. Koska CPT:t on määritetty ”Custom Post Type”-lisäosan kautta voidaan käyttää kolmatta vaihtoehtoa ja yksinkertaisesti käydä napauttamassa lisäosan valikosta kaikkien haluttujen CPT:den kohdalta ”Show in GraphQL”-valinta. Kun asetus kytketään päälle ”Custom Post Type”-lisäosa tulee määrittelemään pakolliset arvot aiemmin annettujen arvojen pohjalta, mikä näkyy kuviosta 26. Tämän vuoksi lisäosaan ei tarvitse tehdä muita muutoksia. Huom. ”Single Name”- ja ”Plural name”-arvoja hyödynnetään, kun määritellään mistä haetaan arvoja ja kuinka paljon.



Kuvio 26. "Custom post type"-lisäosan WPGraphQL-asetukset

Seuraavaksi on hyvä testata, pystyykö GraphQL välittämään CPT:n sisältäviä artikkeleita. Kuviossa 27 on tehty kysely, joka hakee aiemmin määritetyllä ”GraphQL Plural Name”-arvolla ”Claimed Structures”-artikkelityypin artikkelit. Nyt kun GraphQL pystyy hakemaan kaikki artikkelityypin artikkelit, voidaan siirtyä uusien dynaamisien artikkelisivujen tekoon.

Kuvio 27. CPT-artikkelien testihaku

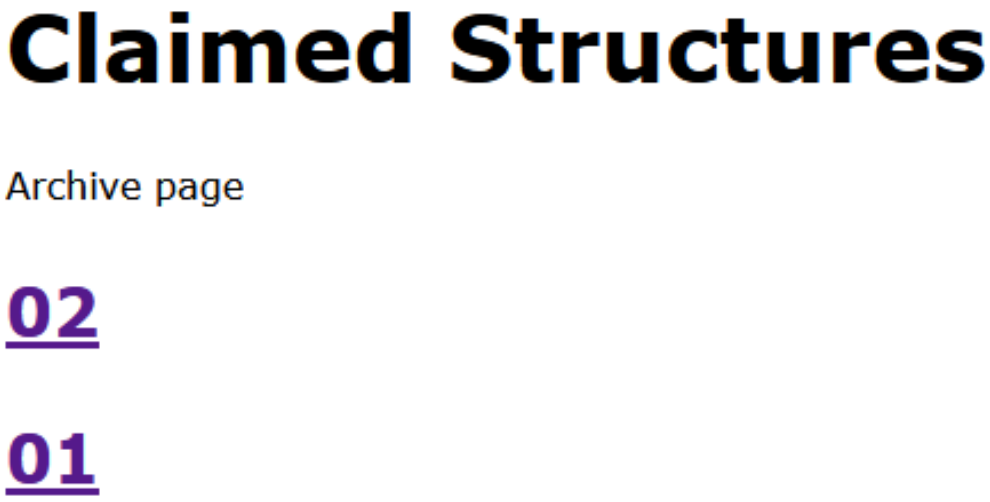
CPT-artikkelisivuja voidaan tehdä monella tavalla. Ensimmäinen tapa tehdä artikkelisivuja on luoda se staattisesti, joka sisältää uuden kansion luomisen ”pages”-kansion juureen, joka sisältää index-tiedoston sekä ”[post].js”-tiedoston. Kyseiset tiedostot toimivat samalla tavalla kuin aiemmin luotu ”posts”-kansion tiedostot, mutta sen sijaan, että kyseiset tiedostot hakisivat kategorioimattomia artikkeleita ne hakevat kuvion 27 mukaisella tavalla tietoja CPT:stä. Staattisesti luodussa artikkelisivussa ei ole mitään vikaan, mutta se rajoittaa kuinka paljon muutoksia WordPressiin voidaan tehdä ennen kuin Next.js-sovellus tulee vaatimaan muutoksia, jonka vuoksi uusia CPT-arkistoja ei voitaisi tehdä ilman, että sovellusta päivitetään.

Dynaamisiin arkistosivuihin ei perehdytä tämän syvemmin, mutta on hyvä tietää, että se on yksi mahdollinen tapa tehdä arkistosivuja ja joissakin tapauksissa se on parempi lähestymistapa. Tässä tapauksessa tehdään staattinen arkistosivu eli sovellukseen tullaan lisäämään uusi kansio, joka nimetään CPT:n mukaan esim. ”claimed-structures”, joka sisältää ”[post].js”- ja ”index.js”-tiedostot. Kuten nimestä voi päätellä ”[post].js”-tiedosto sisältää templaatin artikkelia varten ja ”index.js”-tiedosto sisältää arkiston. Ainoat eroavaisuudet kansioiden ”posts” ja ”claimed-structures” tiedostojen välillä löytyy uri-osoitteen laatimisessa ja minkä funktion kautta ”[post].js” ja index tiedostot saavat tietonsa, kuten kuviossa 28 näytetään. Kuviossa 27 on esitelty miten CPT:stä saadaan haluttu tieto navigoimalla aiemmin määriteltyyn olioon GraphQL-kyselyiden sisällä. Jotta tiedostoon voidaan lisätä sisältöä WordPressin kautta kuviossa 28 näkyy, että tiedostoon on lisätty ”getPage”-funktio, jolla haetaan ”claimed-structures”-nimiseltä sivulta sisältöä. Tämä myös parantaa WordPressin puolen hallinnointia, kun admin-näkymässä näkee kaikki sivut, jotka ovat käytössä Next.js-tiedostossa.

```
import Posts from '../../src/components/posts'

import { getAllPostsFromClaimedStructures, getPage } from '../api/graphql'

// URL: localhost:3000/claimed-structure
export default function ClaimedStructure({ posts, page }) {
  console.log(page.page.title)
  return (
    <div>
      <main>
        <h1>{ page.page.title }</h1>
        <div dangerouslySetInnerHTML={{ __html: page.page.content }} />

        <Posts posts={ posts } currentURI='/claimed-structures/' />
      </main>
    </div>
  )
}

export const getStaticProps = async () => {
  const page = await getPage('claimed-structures')
  const posts = await getAllPostsFromClaimedStructures()
  return {
    props: {
      page: page,
      posts
    }
  }
}
```

Kuvio 28. "Index.js"-tiedoston eroavaisuudet

Näiden muutoksien jälkeen sovelluksesta pitäisi löytyä ”localhost:3000/cpt-name/”-osoite, jossa on listattu kaikki CPT:n sisältämät artikkelit, kuten kuviossa 29 näkyy. Artikkeleita painamalla käyttäjän pitäisi päätyö artikkelisivulle, joka käyttää ”localhost:3000/cpt-name/article-name”-osoitetta. Muut arkistosivut voidaan luoda kopioimalla kansion sisältö, muokkaamalla kyselyt hakemaan tiedot toisesta CPT:stä ja määrittelemällä uusi sivu WordPressissä, joka toimii arkistosivuna.



Kuvio 29. “Claimed Structures”-arkistosivu

Navigaatiovalikko

Tällä hetkellä sovelluksessa on erittäin vaikea navigoida muille sivuille, jonka voi ratkaista lisäämällä navigaatiovalikon. Jotta sovellus olisi helposti muokattava sen tulisi hyödyntää mahdollisimman monta WordPressiin rakennettua ominaisuutta. WordPressissä voidaan helposti muokata ja luoda uusia navigaatiovalikoita käyttäen ulkoasun alta löytyvää valikot näkymää, jonka kautta voidaan hallinnoida navigaatiovalikoita ”drag & drop”-menetelmällä.

Navigaatiovalikon tekeminen tulee vaatimaan yhden uuden komponentin ja yhden uuden GraphQL-kyselyn. Riippuen miten valikkoa halutaan käyttää, kysely tulee lisätä, joko komponentin sisään tai sen ulkopuolelle. Jos valikon halutaan muuttavan sisältöään sen perusteella millä sivulla

kyseinen valikko on, voidaan kysely lisätä komponentin ulkopuolelle, mutta jos halutaan ettei valikon sisältö muutu ulkopuolisten tekijöiden mukaan kysely voidaan lisätä itse komponenttiin. Tässä kehitystyössä kysely tulee komponentin ulkopuolelta komponenttiin. Kuviossa 30 näkyy miten ”navigation"-komponentti käy listan läpi ja luo linkit siitä löytyvien olioiden perusteella. Komponentista löytyy myös yksi ehto, joka varmistaa, ettei komponenttiin luoda ylimääräistä linkkiä ”Front page”. Kyseistä linkkiä ei luoda, koska sen sisältämä osoite on virheellinen, minkä vuoksi kyseinen linkki pitäisi luoda staattisesti sovelluksessa tai toisena vaihtoehtona map-metodin sisältämään ehtoon voisi lisätä uuden templaatin, jota käytetään kun navigaatiovalikko sisältää ”Front page”-linkin.

```
import Link from "next/link"

export default function Navigation({ WPLinks=[] }) {
  const menuItems = WPLinks[0].menuItems.edges

  return (
    <nav>
      <ul>
      {
      menuItems.map((item, index) => (
        // Excluding unwanted links
        (item.node.label !== 'Front page') ?
          <li key={index}>
            <Link href={item.node.path}>
              <a>{item.node.label}</a>
            </Link>
          </li>
        : ''
      ))
      }
      </ul>
    </nav>
  )
}
```

Kuvio 30. Navigaatiovalikko-komponentti

Komponentin lisäksi sovellukseen on luotu uusi GraphQL-kysely, joka hakee argumentin mukaisen navigaatiovalikon WordPressistä. Kuviossa 31 on kuvattu, miten kyseinen kysely on tehty hyödyntäen aiempia kyselyitä. Tämän myötä kyselyssä ei pitäisi olla tässä vaiheessa mitään uutta syntaksin suhteen.

```
// Fetch preset navigation
export const fetchNavigationMenu = async (slug='main-navigation') => {
  const data = await fetchAPI(`
    query MenuBySlug($slug: String!) {
      menus(where: {slug: $slug}) {
        nodes {
          slug
          menuItems {
            edges {
              node {
                label
                path
              }
            }
          }
        }
      }
    }
    `,
    {
      variables: {
        slug: slug
      }
    }
  )
  return data?.menus?.nodes
}
```

Kuvio 31. Navigaatiovalikon GraphQL-kysely

Komponentin ja kyselyn luonnin jälkeen sovelluksesta löytyy kuvion 32 mukainen yksinkertainen valikko, jonka sisältöön voi vaikuttaa ulkopuoliset tekijät. Tässä vaiheessa sovelluksessa pitäisi olla kaikki pääominaisuudet, joiden varaan voidaan tehdä lopullinen sovellus hyödyntäen WordPres-sistä löytyviä sisällönhallinta ominaisuuksia, muokkaamalla templaatteja, jotta kaikki halutut tiedot näkyvät tarkoitetulla tavalla ja viimeiseksi sovellus tulee tarvitsemaan uuden ulkoasun paranta-maan käyttäjäkokemusta.

Kuvio 32. Lopullinen navigaatiovalikko-komponentti

7 Pohdinta

Tässä luvussa pohditaan ja tarkastellaan, onko tutkimuksen tavoitteet saavutettu. Luvussa vastataan seuraaviin kysymyksiin: onko opinnäytetyön tavoitteet saavutettu, minkälaisia tuloksia saatiin, miten työssä on onnistuttu, missä on epäonnistuttu, miten luotettavaa työn sisältö on, miten tuloksia voidaan hyödyntää, kuinka eettinen tutkimus on ja miten toteutusta voidaan jatkokehittää.

7.1 Tavoitteet ja tulokset

Opinnäytetyössä oli tavoitteena tutustua headless-toteutuksen hyötyihin ja haittoihin sekä selvittää miten toimiva headless-toteutus voidaan toteuttaa käyttämällä backend-ratkaisuna perinteistä WordPress-sisällönhallintajärjestelmää ja frontend-ratkaisuna Next.js-frameworkiä. Työssä oli tarkoitus selvittää missä tapauksissa headless-toteutus voisi olla haitallinen ja miten headless-toteutus voidaan toteuttaa käyttämällä backend-teknologiaa, joka on laajasti käytössä, mikä ei normaalisti tue headless-toteutusta.

Opinnäytetyön teoriaosuudessa onnistutaan antamaan useita näkökulmia miksi headless-toteutus saattaa olla hyvä lähestymistapa sovelluksissa. Headless-toteutuksien luonteen vuoksi opinnäytetyössä ei kuitenkaan onnistuttu paikantamaan tilanteita, joissa headless-toteutukset olisivat pakollisia tai haitallisia, jonka vuoksi opinnäytetyössä tarkasteltiin case-tutkimuksia, joiden avulla pyrittiin antamaan konkreettisia esimerkkejä tilanteista, joissa headless-toteutuksella on ollut positiivisia vaikutuksia. Case-tutkimuksien yhteydessä käytiin myös läpi syitä, miksi tutkimuksien yritykset halusivat kehittää olemassa olevaa toteutusta tai vaihtaa kokonaan toteutustapaa.

Opinnäytetyön empiirisessä osuudessa onnistuttiin luomaan toimiva headless-toteutus käyttämällä aiemmin mainittuja teknologioita. Toteutus sisältää yksinkertaisen Next.js-sovelluksen, jota voidaan käyttää pohjana headless-toteutuksissa. Vaikka työssä onnistuttiin luomaan toimiva toteutus se ei tarkoita, etteikö toteutuksessa olisi parannettavaa. Jos sovellusta halutaan käyttää pohjana useamassa projektissa, sitä pitäisi jatkokehittää. Jatkokehityksen myötä toteutusta voidaan käyttää alustana ilman, että joitakin ominaisuuksia jouduttaisiin tekemään uudelleen.

Next.js-sovelluksen tavoitteena oli korvata WordPress-teema frontend-teknologialla, jolla voidaan tuoda uusia ominaisuuksia käyttämällä moderneja sovelluskehityksen ominaisuuksia säilyttämällä kuitenkin vanhan sivurakenteen. Sovelluksen rakenne täsmää WordPressin ”Twenty Twenty-One”-teemassa esiintyvää rakennetta ja sisältää monia samoja ominaisuuksia kuin kyseinen teema, mutta sovelluksesta puuttuu muutamia yleisiä ominaisuuksia, kuten dynaaminen CPT-arkistosivujen luonti ja arkistosivun pagination-valikko. Dynaaminen CPT-arkistosivun luonti ei ole jokaisella verkkosivulla oleellinen ominaisuus, jonka vuoksi ominaisuuden lisäystä ei koettu oleelliseksi. Toisin kuin dynaaminen arkistosivujen luonti, pagination-valikko on monessa sovelluksessa käytetty valikko, mikä voidaan toteuttaa monella eri tavalla riippuen mikä soveltuu sovelluksen käyttötarkoitukseen. Pagination on ominaisuus, mikä pitäisi lisätä jatkokehityksen yhteydessä, mikäli pohjaa halutaan hyödyntää myös muissa projekteissa.

7.2 Eettinen arviointi

Opinnäytetyössä ei käsitellä kyselyistä tuotettua tietoa tai yksilöiviä henkilötietoja eikä se käsittele arkoja aiheita. Työn tarkoituksena on antaa parempi kokonaiskuva työn aiheesta ja minkälaisissa tilanteissa se voisi olla hyödyllinen. Työssä jaetaan WordPress-ympäristössä toimiva toteutustapa, joka pohjautuu internetistä löytyvään tietoon.

7.3 Luotettavuuden arviointi

Opinnäytetyössä pyrittiin käyttämään mahdollisimman tuoretta ja todeksi todettua tietoa luotettavista lähteistä. Työssä ei kuitenkaan aina ollut mahdollista käyttää tuoreinta lähdettä, eivätkä edellä mainitut kriteerit välttämättä takaa sitä, etteikö työ sisältäisi virheellistä tietoa. Lähteestä saatua tietoa voidaan tulkita monella eri tavalla, minkä myötä useat henkilöt voivat päätyä erilaisiin johtopäätöksiin lähteiden sisältämistä tiedoista. Tämä vaikuttaa myös työn luotettavuuteen, jos työssä on virheellisesti tulkittu lähteen tietoja. Työssä käytetään monenlaisia lähteitä, joista epäluotettavimpia ovat artikkelit, jotka on kirjoittanut yksityishenkilöt. Yksityishenkilöiden taustaa on usein vaikea tutkia, koska henkilö voi jakaa nimensä monen muun henkilön kanssa, saatava tieto henkilöstä voi olla virheellistä tai jossain tapauksissa henkilöstä ei löydy mitään tietoa. Asiantuntijoihinkaan ei voi aina luottaa, koska heillä voi olla huonoja kokemuksia tai agenda, joka vaikuttaa miten aihe on selitetty tai miten he näkevät kyseisen aiheen, jonka myötä tutkimus voi sisältää tietoa, joka voi antaa vääristyneen kuvan tietystä aiheesta.

7.4 Jatkokehitys

Jatkokehityksenä Next.js-sovellukseen voisi lisätä aiemmin mainitut dynaamisen artikkelisivun luonti ja pagination-valikon ominaisuudet. GraphQL-kyselyitä voisi kehittää hyödyntäen GraphQL-kyselykielen edistyneitä konsepteja, joiden avulla voidaan saavuttaa muun muassa täsmällisempiä ja uudelleenkäytettäviä kyselyitä. Näiden lisäksi Next.js-sovelluksen templaatteja voisi kehittää, jotta niitä olisi helpompi käyttää muilla samankaltaisilla sivuilla. Tämä myös vaikuttaisi sovelluksen samanlaisuuteen ja ylläpidettävyyteen sovelluksen kasvaessa.

Opinnäytetyön aihe koskee nopeasti edistyvää teknologiaa, josta voisi teettää uuden tutkimuksen myöhemmin, jossa voitaisiin käsitellä, miten headless-sovelluksien kehitys on muuttunut ja onko se edelleen relevantti tapa kehittää sovelluksia esimerkiksi vuonna 2025. Tämä opinnäytetyö raapaisi vasta headless-toteutuksen pintaa, missä käsitteltiin vain frontend-sovelluksen toteuttamiseen liittyviä olennaisia asioita. Tutkimuksessa ei muun muassa käsitelty, miten headless-toteutuksen backend voitaisiin suunnitella ja toteuttaa, jos haluttaisiin käyttää headless-teknologioita backendin puolella.

Lähteet

Angerer, D. N.d. Headless CMS explained in 5 effective minutes, verkkosivusto. Linz: Storyblok. Viitattu 17.2.2022. https://www.storyblok.com/tp/headless-cms-explained.

Baranowski, W. 2021. React JS: Advantages and Disadvantages, verkkosivusto. Katowice: Massive Pixel Creation. Viitattu 16.2.2022. https://massivepixel.io/blog/react-advantages-disadvantages/#Problems_with_SEO.

Create a New React App. N.d. Kehittäjille suunnattu dokumentaatio React-kirjastosta, verkkosivusto. Menlo Park: Meta Platforms. Viitattu 15.3.2022. https://reactjs.org/docs/create-a-new-react-app.html.

Custom Taxonomies. N.d. WPGraphQL-lisäosan dokumentaatio, verkkosivusto. Lontoo: WPEngine. Viitattu 10.4.2022. https://master--wpgraphql-docs.netlify.app/getting-started/custom-taxonomies/.

Documentation. N.d. Kehittäjille suunnattu dokumentaatiosivusto ACF-lisäosalle. Delicious Brains. Viitattu 4.4.2022. https://www.advancedcustomfields.com/resources/.

Frostigon. 2022. Frostigon-yhtiön verkkosivuston etusivu. Jyväskylä: Frostigon. Viitattu 18.3.2022. https://frostigon.com/.

GetStaticPaths. N.d. Kehittäjille suunnattu Next.js-dokumentaatio, verkkosivusto. Walnut: Vercel. Viitattu 7.4.2022. https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#fallback-pages.

GetStaticProps. N.d. Kehittäjille suunnattu Next.js-dokumentaatio, verkkosivusto. Walnut: Vercel. Viitattu 6.4.2022. https://nextjs.org/docs/basic-features/data-fetching/get-static-props.

Getting Started. N.d. Vercel-yhtiön ylläpitämä Next.js-dokumentaatio, verkkosivusto. Walnut: Vercel. Viitattu 15.3.2022. https://nextjs.org/docs.

Google Trends. 2022. Googlen tarjoama palvelu, jonka kautta voi tarkastella hakutuloksien suosiota, verkkosivusto. Mountain View: Google. Viitattu 5.3.2022. https://trends.google.com/trends/explore?date=2013-01-01%202022-01-01&q=Headless%20CMS.

Gopalakrishnan, A. 2021. GraphQL vs REST API, verkkosivusto. Wroclaw: Benjamas. Viitattu 18.2.2022. https://bejamas.io/blog/graphql-vs-rest-api/.

Goyal, Y. N.d. Difference between static vs dynamic web page, verkkosivusto. Mumbai: Educba. Viitattu 10.3.2022. https://www.educba.com/static-vs-dynamic-web-page/.

Gupta, L. 2021. REST API – N+1 Problem, verkkosivusto. RESTful API. Viitattu 15.3.2022. https://restfulapi.net/rest-api-n-1-problem/.

Huss, N. 2022. 7 Best WordPress WYSIWYG Editors Reviewed & Compared, verkkosivusto. Lontoo: Siteefy. Viitattu 17.3.2022. https://siteefy.com/wordpress-wysiwyg-editors/.

Ieva, T. 2021 GraphQL vs. REST: Which is the best for API Development, verkkosivusto. Kaunas: Hostinger. Viitattu 18.2.2022. https://www.hostinger.com/blog/graphql-rest-best-for-api-development.

Jokinen, A. N.d. Laadullisen tutkimuksen näkökulmat, verkkosivusto. Tietoarkisto. Tampere: Yhteiskuntatieteellinen tietoarkisto. Viitattu 26.3.2022. https://www.fsd.tuni.fi/fi/palvelut/menetelmaopetus/kvali/mita-on-laadullinen-tutkimus/laadullisen-tutkimuksen-nakokulmat/.

Juhila, K. N.d. Laadullisen tutkimuksen ominaispiirteet, verkkosivusto. Tietoarkisto. Tampere: Yhteiskuntatieteellinen tietoarkisto. Viitattu 26.3.2022. https://www.fsd.tuni.fi/fi/palvelut/menetelmaopetus/kvali/mita-on-laadullinen-tutkimus/laadullisen-tutkimuksen-ominaispiirteet/.

Kendal, R. 2020. Using WordPress as a headless CMS with Next.js, verkkosivusto. Dev Community. Viitattu 9.4.2020. https://dev.to/kendalmintcode/using-wordpress-as-a-headless-cms-with-next-js-2h5p.

Lisäosat. N.d. WordPress-lisäosakirjasto, verkkosivusto. WordPress Foundation. Viitattu 15.2.2022. https://fi.wordpress.org/plugins/.

Mohan, M. 2020. Why you should learn Next.js as a React developer, verkkosivusto. San Francisco: Free Code Camp. Viitattu 16.2.2022. https://www.freecodecamp.org/news/why-you-should-learn-next-js-as-a-react-developer/.

Morris, W. 2022. A quick guide to headless WordPress, verkkosivusto. Kaunas: Hostinger. Viitattu 20.2.2022. https://www.hostinger.com/tutorials/headless-wordpress.

Niżyński, D. 2020. What is Next JS and why should you use it in 2021, verkkosivusto. Białystok: Pagepro. Viitattu 16.2.2022. https://pagepro.co/blog/what-is-nextjs/.

OneBlade Increases Refill Subscription Rate By 579%. N.d. Shogun-yhtiön teettämä case-tutkimus OneBlade-yhtiön headless-toteutuksesta, verkkosivusto. Walnut: Shogun Labs. Viitattu 17.3.2022. https://getshogun.com/headless-case-study/oneblade.

Rascia, T. 2021. An Introduction to GraphQL, verkkosivusto. New York: DigitalOcean. Viitattu 15.3.2022. https://www.digitalocean.com/community/tutorials/an-introduction-to-graphql.

REST APIs. 2021. IBM:n ylläpitämä REST API artikkeli, verkkosivusto. North Castle: IBM. Viitattu 18.02.2022. https://www.ibm.com/cloud/learn/rest-apis.

Singh, J. 2020. Headless architecture. Why it’s becoming the new normal, verkkosivusto. Dublin: Accenture. Viitattu 17.2.2022. https://www.accenture.com/nl-en/blogs/insights/headless-architecture-why-its-becoming-the-new-normal.

Smal, A. 2022. Miksi Frostigonin asiakas on päätynyt headless-toteutukseen, suullinen tietolähde. Jyväskylä: Frostigon. Viitattu 22.3.2022.

Support. 2016. WordPress-kehittäjille suunnattu verkkosivusto, joka sisältää mm. päivityksien sisältämät muutokset. WordPress Foundation. Viitattu 16.2.2022. https://wordpress.org/support/wordpress-version/version-4-7/.

Static vs Dynamic Website. 2021. Artikkeli staattisien ja dynaamisien sivustojen erosta, verkkosivusto. Noida: Geeks for Geeks. Viitattu 17.3.2022. https://www.geeksforgeeks.org/static-vs-dynamic-website/.

Taxonomies. N.d. Kehittäjille suunnattu WordPress-dokumentaatio artikkeleiden ryhmittelyyn, verkkosivusto. WordPress Foundation. Viitattu 4.4.2022. https://wordpress.org/support/article/taxonomies/.

Traditional CMS vs Headless CMS. 2021. Artikkeli perinteisien ja headless-sisällönhallintajärjestelmien välisistä eroista, verkkosivusto. Richmond: UDig. Viitattu 10.3.2022. https://www.udig.com/digging-in/traditional-cms-vs-headless-cms/.

Umbraco helped Skoringen tie their laces. N.d. Umbraco-yhtiön teettämä case-tutkimus Skoringe-yhtiön headless-toteutuksesta, verkkosivusto. Odense: Umbraco. Viitattu 17.3.2022. https://umbraco.com/case-studies-testimonials/skoringe.

Why React uses className over class attribute. 2021. Blogiartikkeli React-kirjaston JSX-koodissa esiintyvistä eroista, verkkosivusto. Noida: Geeks for Geeks. Viitattu 13.4.2022. https://www.geeksforgeeks.org/why-react-uses-classname-over-class-attribute/.

WordPress statistics. 2022. Manaferra-yhtiön teettämä selvitys WordPressin suosiosta, verkkosivusto. Pristina: Manaferra. Viitattu 28.3.2022. https://www.manaferra.com/wordpress-statistics/.

WPGraphQL for Advanced Custom Fields. N.d. WPGraphQL-lisäosan dokumentaatio verkkosivusto. Lontoo: WPEngine. Viitattu 11.4.2022. https://www.wpgraphql.com/acf/.