Tampereen ammattikorkeakoulu

Tyyppiturvallisuus FullStack TypeScript -sovelluksessa

Joona Rämö

OPINNÄYTETYÖ
Marraskuu 2023

Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

RÄMÖ, JOONA:
Tyyppiturvallisuus FullStack TypeScript -sovelluksessa

Opinnäytetyö 30 sivua
Marraskuu 2023

Tässä opinnäytetyössä syvennytään tyyppiturvallisuuden merkitykseen ja sen toteutumiseen FullStack TypeScript -sovelluksissa. Työ valottaa tyyppien roolia ohjelmistokehityksessä ja erittelee hyötyjä sekä haasteita staattisesti ja dynaamisesti tyypitetyissä ohjelmointikielissä.

FullStack -kehityksen kontekstissa tarkastellaan, miten TypeScript tukee sekä frontend- että backend-puolen kehitystä ja minkälaisia erityishaasteita tyyppiturvallisuuden varmistamisessa saattaa kohdata, kun järjestelmän eri osat kommunikoivat keskenään.

Työssä kiinnitetään erityistä huomiota monorepon käyttöön esimerkkiprojektissa ja siihen, miten tyyppiturvallisuus voidaan varmistaa sekä sovelluksen frontend- että backend-osuudessa projektin koosta riippumatta. Tyyppiturvallisuus vahvistetaan jakamalla sovelluksessa käytettäviä tyyppejä näiden eri osa-alueiden kesken. Tavoitteena on taata vahva tyypitys ja siitä saatavat hyödyt sekä sovelluksen käännösvaiheessa että myös ajonaikaisesti.

Opinnäytetyö osoittaa, että vaikka tyyppiturvallisuuden saavuttaminen vaatii huolellisuutta ja syvällistä ymmärrystä ohjelmistokehityksen prosesseista, se tarjoaa merkittäviä hyötyjä koodin laatuun, ylläpidettävyyteen sekä virheiden vähentämiseen. Lisäksi se parantaa yleisesti myös kehittäjäkokemusta.

Asiasanat: typescript, tyyppiturvallisuus, full-stack-kehitys

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Business Information Systems
Software Engineering

RÄMÖ, JOONA:
Type Safety in Full-stack TypeScript Applications

Bachelor's thesis 30 pages
November 2023

This thesis focuses on type safety within TypeScript applications, which span both frontend and backend systems. The thesis covers the pros and cons of having static versus dynamic types by exploring the core essence of types in software development.

Based on the study, TypeScript emerges as a powerful tool for developers on both ends: frontend and backend. A significant part of this thesis focuses on the integration of monorepos and the challenges of ensuring consistent type safety, particularly when there is interplay between different components of a system.

One clear conclusion drawn from this study is the importance of type safety. Even though mastering it requires diligence and a deep understanding of how the software works, the rewards in terms of code robustness, easier maintenance, and reduced errors are significant.

Key words: typescript, type safety, full-stack development

SISÄLLYS

ERITYISSANASTO

Boolean	Tietotyyppi ohjelmistossa. Sisältää totuusarvon (tosi/epätosi).
Bugi	Ohjelmistossa esiintyvä virhe tai puute, joka aiheuttaa ohjelman toimimattomuuden tai odottamattoman käyttäytymisen.
DSL	Toimialakohtainen ohjelmointikieli, joka on suunniteltu tietyntyyppisten ongelmien ratkaisemiseen tietyssä toimintaympäristössä.
IDE	Integrated Development Environment. Integroitu kehitysympäristö, joka yhdistää useita kehitykseen liittyviä työkaluja yhteen käyttöliittymään. Tällainen on esimerkiksi Visual Studio Code.
Monorepo	Projektin rakenne, jossa yksi repositorio sisältää projektin eri osia.
ORM	Object-Relational Mapping. Tekniikka, joka mahdollistaa relaatiotietokantojen ja objektipohjaisten ohjelmointikielten välisen suhteen hallinnan.
pnpm	Pakettimanageri Node.js-projekteille, joka on vaihtoehto perinteisemmille npm:lle ja yarnille.
Refaktorointi	Ohjelmistokehityksen käytäntö, jossa parannetaan olemassa olevan koodin rakennetta ilman, että muutetaan sen toiminnallisuutta. Tavoitteena on parantaa koodin selkeyttä ja ylläpidettävyyttä.

1 JOHDANTO

Nykyään ohjelmistokehitys on kasvava ja jatkuvasti kehittyvä ala, jossa kehittäjät etsivät tehokkaita tapoja rakentaa luotettavia ja joustavia sovelluksia. Eräs keskeinen tekijä näiden tavoitteiden saavuttamisessa on tyyppiturvallisuus, joka auttaa varmistamaan, että ohjelmakoodi toimii oikein ja että virheelliset arvot tai operaatiot havaitaan ennen ohjelman suoritusta. Tyyppiturvallisuus on erityisen tärkeää työskennellessä suurten ja monimutkaisten projektien parissa, joissa koodin ymmärtäminen ja ylläpitäminen voi olla haasteellista.

TypeScript on suosittu valinta ohjelmistokehittäjille, jotka haluavat saada kielessä käytetyn vahvan tyypityksen edut, mutta kuitenkin säilyttäen JavaScriptin ympärille rakennetun vuosien varrella kehittyneen ekosysteemin, joka edesauttaa kehittäjää rakentamaan projektiaan sulavammin.

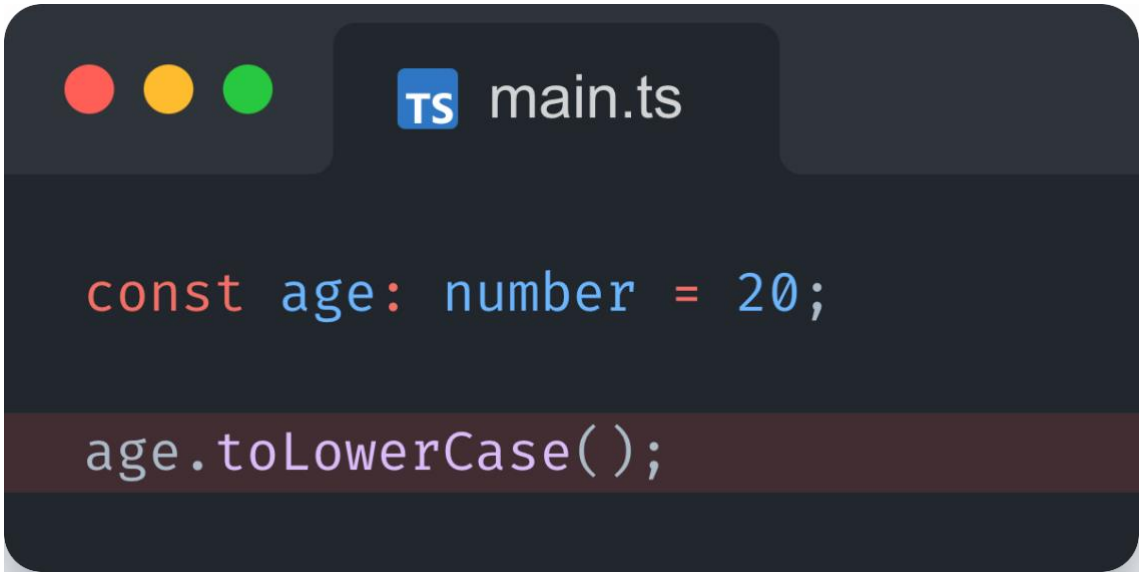
Tämän opinnäytetyön tavoitteena on tarkastella tyyppiturvallisuutta full-stack TypeScript-projekteissa. Työssä selvitetään, mitä tyyppiturvallisuus tarkoittaa, miksi se on tärkeää ohjelmistokehityksessä ja millaisia etuja se voi tarjota. Lisäksi työssä käsitellään eri tapoja saavuttaa tyyppiturvallisuus TypeScript-projekteissa ja miten niitä voidaan hyödyntää käytännössä.

2 TYYPPITURVALLISUUS JA TYYPITYS

2.1 Tyypit ohjelmistokehityksessä

Cardellin (2004) Microsoftille suorittaman tutkimuksen mukaan tyypitysjärjestelmän perustavanlaatuinen tarkoitus on estää suoritusvirheitä ohjelman ajon aikana. Tämän tärkeän tehtävän vuoksi tyypit ovat keskeinen osa ohjelmointikieliä ja ohjelmistokehitystä yleisesti. Ne määrittelevät muuttujien, funktioiden ja arvojen tyyppitiedot, mikä auttaa ymmärtämään koodin rakennetta ja toimintaa. Tyypit eri ohjelmointikielissä voivat hieman vaihdella, mutta useimmiten niihin kuuluvat ainakin numerot, merkkijonot, boolean-arvot, taulukot ja objektit. JavaScriptissä näitä perustyyppejä kutsutaan primitiivisiksi arvoiksi (Ecma International, 2023).

Tyypit ovat tärkeitä ohjelmistokehityksessä, koska ne auttavat välttämään virheitä koodissa ja parantamaan koodin luettavuutta. Kun muuttujan tai funktion tyyppi on määritelty, se varmistaa, että muuttujaa tai funktiota käytetään oikein koodissa. Tämä auttaa välttämään virheitä, jotka voivat aiheuttaa ohjelman kaatumisen tai muun toimintahäiriön.



KUVA 1. Virheellinen funktiokutsu, josta TypeScript varoittaa, sillä numeroa ei voida muuttaa pieniksi kirjaimiksi.

Tyypit myös auttavat ymmärtämään koodin rakennetta ja toimintaa. Kun koodissa on määritelty tyypit, se tekee koodin luettavammaksi ja helpommin ymmärrettäväksi. Tyyppien käyttö auttaa myös kehittäjiä ymmärtämään koodin toimintaa ja tekemään muutoksia helpommin.

Erilaiset ohjelmointikielet käyttävät erilaisia tyyppijärjestelmiä. Joissakin kielissä tyypit ovat vahvasti tarkistettuja, mikä tarkoittaa, että jokaisen muuttujan ja funktion tyyppi on määriteltävä ja tarkistettava. Toisissa kielissä taas tyypit ovat heikommin tarkistettuja, mikä tarkoittaa, että muuttujien ja funktioiden tyyppejä ei tarvitse määritellä ollenkaan, tai niitä voidaan muuttaa koodin suorituksen aikana. Cardellin (2004) mukaan ohjelmointikielen tarkka tyypitys ei kuitenkaan aina takaa sen turvallisuutta, vaan esimerkiksi C-kielessä on sen tyypityksestä huolimatta mahdollista tehdä epäturvallisia operaatioita, jotka saattavat johtaa odottamattomiin virheisiin.

Yhteenvetona, tyypit ovat tärkeä osa ohjelmointia ja ohjelmistokehitystä, koska ne auttavat välttämään virheitä koodissa, parantavat koodin luettavuutta ja auttavat ymmärtämään koodin toimintaa. Erilaiset ohjelmointikielet käyttävät erilaisia tyyppijärjestelmiä, ja niiden käyttö voi vaihdella riippuen kunkin kielen ominaisuuksista ja käyttötarkoituksesta.

2.2 Tyyppiturvallisuus

Tyyppiturvallisuus viittaa ohjelmointikielen tai ohjelmiston ominaisuuteen estää tai minimoida tietotyyppien väärinkäytöksistä aiheutuvat virheet (Cardelli 2004). Tyyppiturvalliset kielet pyrkivät varmistamaan, että ohjelman eri osat käsittelevät vain niille määriteltyjä tietotyyppejä eivätkä käsittele sopimattomia tai odottamattomia arvoja.

Yksi merkittävä tyyppiturvallisuuden etu on virheiden havaitseminen jo kehitysvaiheessa ennen ohjelman suorittamista. Kun ohjelmoijat voivat tunnistaa tietotyyppien käyttöön liittyviä virheitä jo kehityksen aikana, se vähentää suoritusaikaisia virheitä, parantaa ohjelman laatua ja säästää pitkällä tähtäimellä kustannuksia.

Therac-25 sädehoitolaite sekä sen käytössä tapahtuneet lukuisat onnettomuudet ovat surullisenkuuluisa esimerkki siitä, miten ohjelmistovirheet voivat johtaa vakaviin seurauksiin. Vuosina 1985–1987 kyseinen laite aiheutti ohjelmistovirheiden vuoksi kuusi säteily-yliannostusta johtaen syöpäpotilaiden kuolemiin. Virhe aiheutti laitteen antamaan potilaille arvioiden mukaan peräti satakertaisen säteilyannoksen normaalista, kun jo viisikertainen annos voi olla tappava. (Leveson, 2010.)

Tyyppiturvallinen koodi on myös helpommin ylläpidettävissä. Tietotyypit tarjoavat selkeän viestin siitä, mitä dataa ohjelmaelementit odottavat ja käsittelevät. Tämä selkeys helpottaa koodin lukemista ja ymmärtämistä, mikä nopeuttaa uusien kehittäjien perehdyttämistä projektiin.

Tyyppiturvallisuus ei ole ainoastaan virheiden estämisen väline. Se voi myös optimoida ohjelman suorituskykyä. Kun järjestelmä tuntee koodissa käytetyt tietotyypit, se voi tehokkaammin valita sopivat tietorakenteet ja algoritmit. Lisäksi tyyppitiedon ansiosta suoritusaikaisia tyyppitarkistuksia voidaan välttää, mikä nopeuttaa ohjelman toimintaa.

Tyyppiturvallisuutta voidaan pitää myös avaimena parempaan kehittäjäkokemukseen. Se auttaa kehittäjiä ymmärtämään nopeasti koodin rakenteen ja tarkoituksen. Työkalut, kuten koodin automaattinen täydennys ja virheiden korostaminen, hyödyntävät tyyppitietoa ja parantavat ohjelmointiympäristön käytettävyyttä. Kehittäjä voi tyyppitiedon myötä myös löytää ohjelmointivirheitä nopeammin, ja esimerkiksi pienet kirjoitusvirheet jäävät paljon helpommin kiinni.

Vahvan tyypityksen huomioiminen ohjelmistokehityksessä on keskeistä ohjelmistojen laadun, luotettavuuden ja ylläpidettävyyden kannalta. Eri ohjelmointikielet ja työkalut tarjoavat erilaisia tyyppiturvallisuuden tasoja, joten kehittäjän on tärkeää valita ne, jotka parhaiten vastaavat hänen tarpeitaan. Etenkin kriittisten järjestelmien kohdalla vahvasti tyypitetyn kielen valitseminen on enemmän kuin perusteltua. Toisaalta yksinkertaisemman sovelluksen kohdalla niille ei välttämättä ole tarvetta.

2.3 Staattisesti ja dynaamisesti tyypitetyt kielet

Staattisesti ja dynaamisesti tyypitettyjen kielten väliset erottuvat piirteet vaikuttavat merkittävästi niiden käyttöön, suorituskykyyn ja kehittäjäkokemukseen. Oracle kertoo dokumentaatiossaan, että staattisesti tyypitettyjen kielten, kuten Java, kääntäjät tekevät tyyppien tarkistuksen ennen ohjelman suorittamista. Heidän mukaansa tämä auttaa varmistamaan, että koodi käyttää tietotyyppejä oikein, ja mahdolliset tyyppivirheet havaitaan ennen ohjelman ajamista. Oraclen mukaan dynaamisesti tyypitetyissä kielissä, kuten Groovy, tyyppien tarkistus suoritetaan taas ohjelman ajon aikana. Tämä joustavuus mahdollistaa nopeamman koodin kirjoituksen, mutta tyyppivirheet voivat jäädä huomaamatta ennen kuin ohjelmaa suoritetaan ja ne aiheuttavat ongelmia. Lisäksi dynaamisesti tyypitetyissä kielissä on käytettävissä vain ohjelmointikieleen itseensä määritetyt tyypit, ja omien tyyppien määrittäminen ei ole mahdollista. (Oracle, 2015.)

Staattisesti tyypitettyjen kielten kääntäjät voivat myös tuottaa tehokkaampaa konekielistä koodia, koska ne tietävät tietotyypit etukäteen. Tämä etukäteistieto mahdollistaa paremman koodin optimoinnin. Dynaamisesti tyypitettyjen kielten suorituskyky voi olla heikompi, koska jatkuva tyyppitarkistus suorituksen aikana saattaa hidastaa ohjelmaa.

Kehitystyössä joustavuus on keskeistä. Dynaamisissa kielissä tietotyyppien eksplisiittistä määrittelyä ei tarvita, mikä voi tehdä koodista lyhyempää ja helpommin muokattavaa. Staattisissa kielissä, vaikka tietotyyppien määrittäminen voi vaatia enemmän koodia, se voi parantaa koodin selkeyttä.

Kehittäjäkokemus on toinen merkittävä näkökulma. Staattisesti tyypitettyjen kielten avulla kehittäjät voivat hyödyntää IDE:n ominaisuuksia, kuten koodin täydennystä ja virheiden korostusta, koska tietotyypit ovat tiedossa etukäteen. Dynaamisissa kielissä kokemus voi olla erilainen. Vaikka niiden joustavuus voi nopeuttaa koodin kirjoittamista, ne eivät välttämättä tarjoa yhtä kattavia työkaluja, mikä voi tehdä kehittämisestä sekä virheiden havaitsemisesta haastavampaa.

Refaktoroinnin ja ylläpidon kannalta staattisesti tyypitettyjen kielten koodissa virheiden havaitseminen on helpompaa kääntäjän tarkastusten ansiosta. Dynaamisten kielten koodissa tyyppivirheitä voi olla vaikeampi havaita ennen ohjelman suoritusta, mikä voi lisätä ongelmia myöhemmässä vaiheessa. Esimerkiksi staattisesti tyypitetyssä kielessä muuttujan nimen tai tyypin vaihtamisen jälkeen ohjelma ei ollenkaan käänny, ennen kuin koko ohjelmasta on tehty yhteensopiva muutokselle. Dynaamisessa kielessä tällainen virhe puolestaan saattaa jäädä kokonaan huomaamatta, sillä virhe tapahtuu vasta ohjelman suoritusvaiheessa.

Staattisilla ja dynaamisilla kielillä on kumpaisellakin omat vahvuutensa ja heikkoutensa. Valinta niiden välillä riippuu usein sovelluksen vaatimuksista, kehittäjien mieltymyksistä ja projektin tavoitteista.

3 TYPESCRIPT

3.1 TypeScriptin esittely

TypeScript on avoimen lähdekoodin ohjelmointikieli, joka on kehitetty parantamaan JavaScriptin kehityskokemusta ja tuomaan tyypitys JavaScriptiin. Se julkaistiin ensimmäisen kerran vuonna 2012 ja sen on kehittänyt Microsoft. TypeScript on yläyhteensopiva JavaScriptin kanssa, mikä tarkoittaa, että voit käyttää JavaScript-koodia TypeScript-projekteissa ja kääntää TypeScript-koodin suoraan JavaScriptiksi. (TypeScript, n.d.)

TypeScript lisää JavaScriptiin staattisen tyypityksen, joka auttaa kehittäjiä tunnistamaan ja korjaamaan virheitä jo kehitysaikana. Staattinen tyypitys tarjoaa myös muita etuja, kuten parempaa koodin itsestään dokumentaatiota ja tehokkaampaa työskentelyä kehitysympäristöissä, jotka tukevat automaattista täydennystä ja koodin refaktorointia.

TypeScriptin avulla kehittäjät voivat määritellä muuttujien, funktioiden ja luokkien tyypit, mikä auttaa varmistamaan, että ne käyttäytyvät oikein ja yhteensopivasti keskenään. TypeScriptin dokumentaatiosta (TypeScript Handbook, n.d.) selviää, että se sisältää myös lisäominaisuuksia, kuten geneeriset tyypit, sekä rajapinnat, jotka antavat kehittäjille lisää joustavuutta ja työkaluja ohjelmistojen suunnitteluun ja rakentamiseen.

Vaikka TypeScript on suosittu etenkin suurissa ja monimutkaisissa projekteissa, se on hyödyllinen myös pienemmissä projekteissa ja jopa yksittäisille kehittäjille. Sen tarjoamat tyyppiturvallisuuden edut, kuten virheiden varhainen havaitseminen, koodin itsestään dokumentaatio ja parempi kehitysympäristöjen integraatio, voivat tehdä ohjelmistokehityksestä nopeampaa, tehokkaampaa ja virheettömämpää.

TypeScriptin suosio on kasvanut viime vuosina, ja se on noussut yhdeksi suosituimmista ohjelmointikielistä. Sen jatkuva kehittyminen pitää huolen, että trendi tuskin on laskemaan päin. TypeScriptin kasvavasta suosiosta kertoo mm. Stack Overflow’n vuosittain kehittäjille järjestettävän kyselyn tulokset.

Viimeisimmän kyselyn (2023) mukaan kaikista sovelluskehityksessä käytettävistä ohjelmointikielistä TypeScript oli vastaajien mukaan suosituin kieli. Vastaajista lähes 39 prosenttia kertoi käyttävänsä TypeScriptiä. Sen edellä olivat ainoastaan JavaScript sekä Python. Ylöspäin menevää trendiä voidaan osoittaa tarkastelemalla kyselyn aikaisempien vuosien tuloksia. Vuonna 2020 järjestetyssä kyselyssä TypeScriptiä ilmoitti käyttävänsä vain reilu 25 prosenttia vastaajista. Tällöin TypeScriptiä suosituimpia kieliä oli paljon enemmän, JavaScriptin ja Pythonin lisäksi myös Java, C# sekä PHP. TypeScriptin suosio on kuitenkin kasvanut vuosittain keskimäärin noin 4–5 prosenttiyksikköä. Kyselyiden tuloksista voidaan osoittaa, että vastaavanlaiseen kasvuun ei ole yltänyt mikään muu ohjelmointikieli.

JavaScript on jo pitkään ollut suosituin ohjelmointikieli verkkosovellusten kehittämisessä. Sen suosio on johtanut monien tehokkaiden kirjastojen ja kehysten syntymiseen, kuten Svelte, React ja Vue. Nämä kirjastot tukevat nykyisellään myös TypeScriptiä suoraan. Kehittäjä voi usein itse päättää, alustaako uuden projektinsa käyttämään TypeScriptiä vai pelkkää JavaScriptiä. Koska kielet ovat yhteensopivat, TypeScriptiä ei tarvitse ottaa kerralla käyttöön koko koodikannassa. TypeScriptiä käyttöönottaessa olemassa olevaan JavaScript-projektiin, on jopa suositeltavaa ottaa tyypitys käyttöön vähitellen (TypeScript, n.d.).

3.2 TypeScriptin heikkoudet

Opinnäytetyössä on tähän asti lähinnä käyty läpi tyyppiturvallisuuden sekä TypeScriptin hyötyjä sekä etuja. On kuitenkin tärkeää tarkastella myös sen heikkouksia ja rajoituksia, jotta voimme arvioida ohjelmointikielen ominaisuuksia laajemmin ja tarkemmin.

TypeScriptin oppiminen vaatii aikaa ja vaivaa, erityisesti niille, jotka ovat tottuneet dynaamisesti kirjoitettuihin kieliin, kuten JavaScriptiin. TypeScriptin laaja ja monimutkainen tyyppijärjestelmä voi aluksi tuntua pelottavalta, ja sen hallitseminen voi olla aikaa vievää. Tämä voi hidastaa kehitystä projektin alkuvaiheessa ja johtaa virheelliseen tai puutteelliseen tyyppien käyttöön.

Vaikka TypeScript on kehittynyt ja tyyppipäättely on parantunut, on olemassa tilanteita, joissa TypeScript ei pysty päättelemään tyyppiä automaattisesti. Tämä voi johtaa virheellisiin tai heikkoihin tyyppeihin, mikä voi vähentää tyyppiturvallisuutta. Kehittäjien on tällöin määriteltävä tyypit manuaalisesti, mikä voi lisätä koodin monimutkaisuutta ja aiheuttaa lisävaivaa. Heikko tyyppipäättely voi myös johtaa virheisiin, jos kehittäjät eivät ole tarkkoja tyypityksessään.

TypeScriptin yksi haasteista on yhteensopivuus olemassa olevien JavaScript-kirjastojen kanssa. Vaikka useimmille suosituille kirjastoille on olemassa TypeScript-tyyppimäärittelytiedostoja, voi silti olla tilanteita, joissa kehittäjien on itse kirjoitettava tyyppimäärittelyt kolmannen osapuolen kirjastoille. Tämä voi olla aikaa vievää ja vaatia syvällistä ymmärrystä kyseisistä kirjastoista. Lisäksi, jos kirjaston tyyppimäärittelyt ovat vanhentuneita tai virheellisiä, se voi johtaa tyyppiturvallisuuden heikkenemiseen sekä odottamattomiin bugeihin ohjelmistossa. Selatessa JavaScript-kirjastojen pakettienhallintatyökalua npm:ää, huomaa kuitenkin, että nykyään TypeScriptin käytöstä kirjastoja luodessa on muodostumassa alan standardi, ja suuri osa uusista kirjastoista on kirjoitettu sitä apuna käyttäen (npm Inc, n.d.). Vanhempia kirjastoja ei useimmiten ole TypeScriptillä kirjoitettu, mutta niille löytyy usein kolmannen osapuolen kirjoittamia tyyppimäärittelyjä, jolloin kirjaston käyttäjä saa tyypit käyttöönsä asentamalla ne erikseen.

Vaikka TypeScript pyrkii parantamaan tyyppiturvallisuutta, se ei voi tarjota täydellistä tyyppiturvallisuutta. TypeScriptin tyyppijärjestelmä on suunniteltu olemaan joustava ja käytännöllinen, mutta tämä tarkoittaa, että se voi sallia joitain ei-turvallisia toimintoja tai heikkoja tyyppejä. Esimerkiksi, *any*-tyyppi voi heikentää tyyppiturvallisuutta, koska se sallii muuttujan olla minkä tahansa tyyppinen, ja ottaa kielen tarjoaman tyyppitarkastuksen kyseiselle muuttujalle kokonaan pois käytöstä (TypeScript Handbook, n.d.). TypeScriptin dokumentaation mukaan *any*-tyyppiä ei tulisi koskaan käyttää, ellei kehittäjä ole parhaillaan siirtämässä JavaScript-projektia TypeScriptiin.

TypeScriptin tyyppijärjestelmä toimii pääasiassa käännösaikana, ja se auttaa havaitsemaan tyyppivirheitä ennen koodin suorittamista (Jansen, Vane & de Wolff 2016, 50). Kuitenkin, koska TypeScript-koodi kääntyy JavaScriptiksi,

runtime-tarkastukset puuttuvat. Tämä tarkoittaa, että TypeScript ei suojaa virheiltä, jotka johtuvat dynaamisista operaatioista tai arvoista, jotka saadaan suoritusajan aikana, esimerkiksi käyttäjän syötteestä tai ulkoisista API-pyynnöistä. Tämän seurauksena kehittäjien on edelleen käytettävä ajonaikaisia tarkastuksia varmistaakseen, että koodi käsittelee oikein näitä dynaamisia arvoja ja tilanteita (Jansen ym. 2016, 51). Ajonaikaisiin tarkastuksiin voi käyttää esimerkiksi Zod-kirjastoa (Zod, n.d.).

Osa kehittäjistä pitää TypeScriptin oletusarvoisia tyyppimäärityksiä tiettyjen natiivien JavaScript-funktioiden, kuten *JSON.parse*:n kanssa liian vapaamuotoisena (Pocock, 2023). TypeScriptissä kyseinen funktio palauttaa aiemmin mainitun *any*-tyypin. Tämä voi johtaa tilanteisiin, joissa koodi ajetaan ilman virheitä kehitysaikana, mutta aiheuttaa ongelmia tuotannossa. Näiden rajoitteiden takia TypeScriptille on kehitetty tätä ogelmaa ratkaiseva erillinen *ts-reset*-kirjasto. Eräs sen ominaisuuksista on muuttaa edellä mainitun *JSON.parse* funktion palautusarvo *any*-tyypistä *unknown*-tyyppiin (Pocock, 2023). Pocock kertoo, että tämä pakottaa kehittäjät varmistamaan palautettavan datan tyypin ennen sen käyttöä, joko validoimalla sen tai käyttämällä eksplisiittistä tyyppimuunnosta. Vaikka on hyvä, että tällaisten ratkaisujen avulla voidaan välttää TypeScriptissä olevia haasteita, sekä parantaa kielen toiminnallisuuksia, on ironista, että kehittäjien täytyy turvautua ulkopuolisiin kirjastoihin korjatakseen kielen alkuperäisiä oletuksia.

4 FULL-STACK KEHITYS TYPESCRIPTIN AVULLA

4.1 Full-stack kehityksen määritelmä ja komponentit

Full-stack kehitys tarkoittaa ohjelmistokehitystä, jossa kehittäjä hallitsee sekä sovelluksen käyttäjälle näkyvän osuuden (frontend), että taustalla pyörivän osuuden (fbackend) teknologiat (MongoDB, n.d.). Tässä kappaleessa käsitellään full-stack kehityksen määritelmää ja komponentteja, keskittyen erityisesti TypeScript-projekteihin.

MongoDB kertoo verkkoartikkelissaan, että full-stack kehittäjä työskentelee sovelluksen kaikilla osa-alueilla, mukaan lukien käyttöliittymän suunnittelu, datan hallinta, palvelinpuolen logiikka ja tietokantojen hallinta. Heidän mukaansa ykyään myös pilvi-infrastruktuurien hallinta kuuluu usein full-stack kehittäjän ominaisuuksiin. Tämä tarkoittaa, että full-stack kehittäjän tulee hallita erittäin laaja valikoima teknologioita ja työkaluja. (MongoDB, n.d.)

4.2 Frontend-kehitys

MongoDB:n verkkoartikkelin mukaan frontend tarkoittaa sovelluksen käyttöliittymää, jonka sen loppukäyttäjät näkevät ja käyttävät. Frontend-kehityksessä keskitytään käyttöliittymän suunnitteluun, interaktioihin ja suorituskykyyn. TypeScript sopii hyvin frontend-kehitykseen, koska se tarjoaa vahvemman tyypityksen ja paremman työkalutuen verrattuna tavalliseen JavaScriptiin. Tämä voi parantaa sovelluksen luotettavuutta ja helpottaa koodin ylläpitoa. Koska selaimet eivät kuitenkaan suoraan tue TypeScriptiä, ei sitä voida selainympäristössä täysin sellaisenaan käyttää. Usein frontend-kehityksessä käytetään jotakin sovelluskehystä, kuten Angular, React tai Svelte.

4.3 Backend-kehitys

Backend-kehityksen MongoDB taasen määrittelee siten, että sen tehtävänä on luoda sovelluksen palvelinpuolen logiikkaa, joka käsittelee tiedon vastaanottamista, käsittelyä ja tallentamista. Backend-kehityksessä keskitytään sovelluksen toiminnallisuuteen, tietoturvaan ja skaalautuvuuteen. TypeScript on erinomainen valinta myös backend-kehitykseen, koska sen tyyppiturvallisuus ja tehokas työkalutuki auttavat ehkäisemään virheitä ja parantamaan koodin ylläpidettävyyttä. Node.js on suosittu palvelinpuolen ympäristö, joka tukee TypeScriptiä ja mahdollistaa sen käytön backend-kehityksessä.

4.4 Tietokantojen hallinta

Tietokantojen hallinta on olennainen osa full-stack kehitystä, koska se mahdollistaa tiedon tallentamisen, haun ja muokkaamisen. Tietokantojen kanssa työskentely voi olla haastavaa, mutta TypeScriptin avulla on mahdollista luoda tyyppiturvallisia ja modulaarisia tietokantaratkaisuja. Node.js:llä, jolla on mahdollista luoda palvelinpuolen ohjelmia TypeScriptiä hyödyntäen, voi käyttää erilaisia tietokantakirjastoja ja ORM-työkaluja kuten Prisma, Sequelize ja TypeORM (Prisma Data, Inc, 2022). Prisman verkkosivujen mukaan ne mahdollistavat mm. tietokantataulujen kenttien tyypittämisen TypeScriptille yhteensopiviksi.

4.5 Yhteys frontendin ja backendin välillä

Full-stack kehityksessä on tärkeää ylläpitää sujuva ja tehokas yhteys frontendin ja backendin välillä. Tämä voidaan toteuttaa erilaisten rajapintojen, kuten REST API:n, GraphQL:n, tai tRPC:n avulla (Prisma Data, Inc, 2023). TypeScript tukee näitä teknologioita ja auttaa rakentamaan tyyppiturvallisia ja selkeitä rajapintoja, joiden avulla frontend ja backend voivat kommunikoida keskenään. Tässä opinnäytetyössä demonstroidaan tarkemmin REST API:n tyyppiturvallista kommunikointia frontendin kanssa käyttäen Prisma-työkalua.

5 TYYPPITURVALLISUUDEN TOTEUTTAMINEN

Tässä luvussa keskitymme tyyppiturvallisuuden toteuttamiseen full-stack TypeScript-projektissa käytännön esimerkin avulla. Esimerkkiprojektissamme hyödynnämme pnpm workspacea monorepon rakentamiseen, mikä mahdollistaa projektin eri osien, kuten backendin ja frontendin, tehokkaan hallinnan ja yhteisen koodin jakamisen (pnpm, n.d.). Tämä luo vahvan pohjan tyyppiturvallisuuden varmistamiselle koko projektissa.

Koska tavoitteenamme on varmistaa tyyppiturvallisuus TypeScript-sovelluksen backendin ja frontendin välillä, on tärkeää, että nämä kaksi osaa sijaitsevat samassa repositoriossa. Tämä mahdollistaa yhteisten tyyppien ja rajapintojen jakamisen kahden osan välillä, mikä edistää koodin ylläpidettävyyttä, vähentää virheiden mahdollisuutta ja sitä myötä etenkin suuremmissa projekteissa parantaa kehitystiimin tuottavuutta.

5.1 Monorepon perusteet

Monorepo on yksi tapa organisoida useita projekteja tai projektin osia yhdessä repositoriossa. Tämä lähestymistapa tarjoaa monia etuja, kuten koodin uudelleenkäytön ja helpon koodin jakamisen projektin osien välillä. Se auttaa myös parantamaan versionhallintaa, sillä kaikki projektin osat säilytetään samassa paikassa, jolloin ne voidaan helpommin pitää synkronoituna. Monorepo voi erityisesti parantaa tyyppiturvallisuutta full-stack TypeScript-projekteissa, koska se mahdollistaa yhteisten tyyppien ja rajapintojen jakamisen backendin ja frontendin välillä.

Esimerkkiprojektissamme käytämme pnpm workspaces -ominaisuutta monorepon hallintaan. pnpm on tehokas pakettimanageri, joka tarjoaa paremman suorituskyvyn ja pienemmän levytilan kulutuksen verrattuna muihin pakettimanagereihin, kuten npm ja yarn. pnpm:n workspaces-ominaisuus on suunniteltu helpottamaan monorepon hallintaa ja nopeuttamaan kehitysprosessia. (pnpm, n.d.)

5.2 Monorepon käyttöönotto

Monorepon käyttöönotto on erityisen hyödyllistä full-stack TypeScript-projekteissa, sillä se helpottaa tyyppiturvallisuuden toteuttamista koko sovelluksen tasolla, sekä frontend- että backend-puolella. Tämä lähestymistapa auttaa myös ylläpitämään selkeän projektirakenteen ja nopeuttaa kehitystä yhtenäistämällä työkaluja ja konfiguraatioita.

Esimerkkisovelluksemme monorepo-rakenne on seuraavanlainen:

KUVA 2. Monorepo-projektin rakenne

Tässä rakenteessa on kolme pääosaa: api, web ja database. Api-kansio sisältää backend-puolen sovelluslogiikan ja palvelimet, kun taas web-kansio sisältää frontend-puolen sovelluksen. Database-paketti sisältää tietokantaskeeman määrittelyä Prisma-kirjastolla, joka mm. osaa generoida tyypit tietokantarakenteelle TypeScriptiä varten. Database-pakettia voidaan hyödyntää sekä sovelluksen frontendissä (web-kansio) että backendissä (api-kansio).

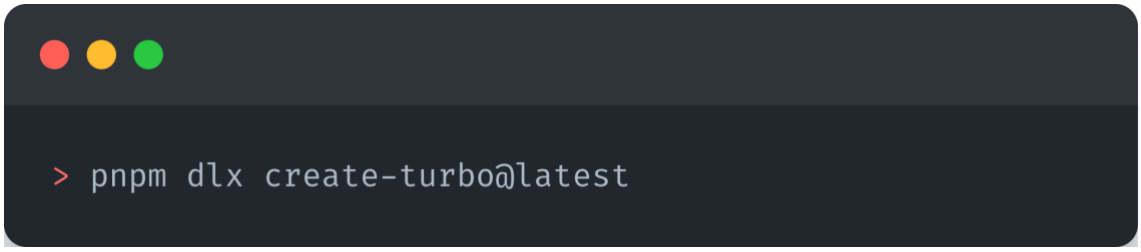
Kun monorepo on otettu käyttöön esimerkkisovelluksessamme, voimme hyödyntää pnpm workspaces -ominaisuutta, joka mahdollistaa yhteisten riippuvuuksien hallinnan, työkalujen jakamisen ja koodin uudelleenkäytön eri projektin osien välillä. Tämä tekee yhteisen tyyppiturvallisuuden toteuttamisesta entistä helpompaa ja johdonmukaista.

Kun sovelluksen eri osat on jaettu monorepossa selkeästi, voimme asentaa ja hallita riippuvuuksia projektin eri osille tehokkaasti. Esimerkiksi, voimme käyttää yhtenäisiä TypeScript-konfiguraatioita ja yhteisiä tyyppikirjastoja, mikä edistää tyyppiturvallisuutta ja ylläpidettävyyttä.

Yhteisen tyyppimäärityksen ansiosta voimme käyttää samoja tyyppejä ja rajapintoja sekä frontend- että backend-puolella, mikä vähentää virheiden mahdollisuutta ja tehostaa kehitysprosessia. Tämä on erityisen hyödyllistä full-stack TypeScript-projekteissa, joissa tyyppiturvallisuus on keskeinen osa koodin laadun varmistamista.

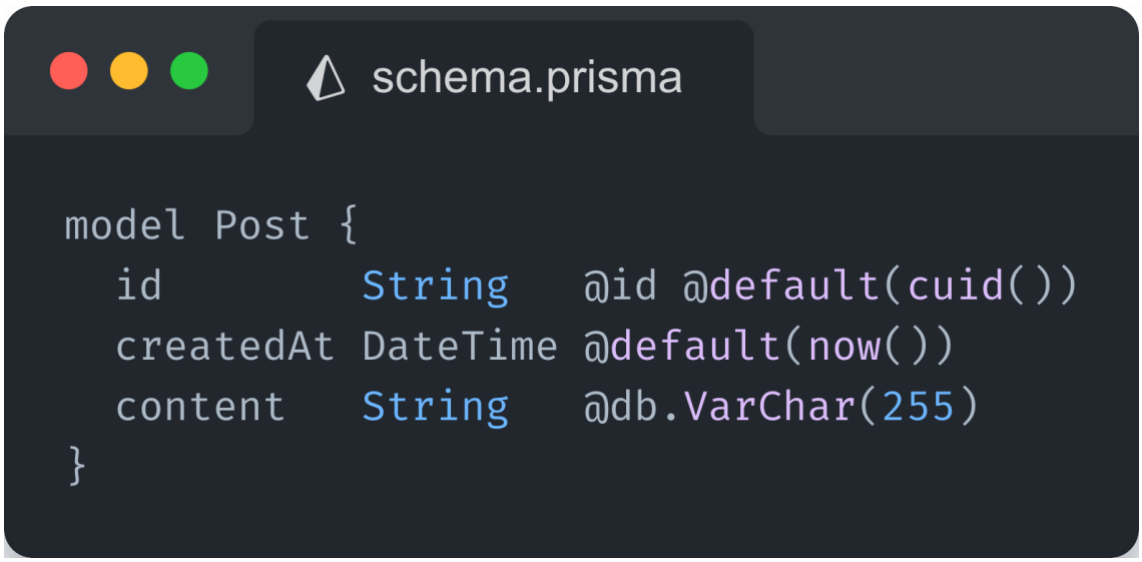
Lisäksi monorepo helpottaa yhteistyötä ja kommunikointia kehitystiimin kesken. Kehittäjät voivat jakaa helposti tietoa ja muutoksia projektin osien välillä, mikä nopeuttaa kehitystä ja auttaa välttämään päällekkäistä työtä.

Monorepon käyttöönotto uutta projektia luotaessa on tehty mahdollisimman helpoksi useiden eri käännösjärjestelmien toimesta. Eräs tällaisista on Turborepo, joka luo kehittäjän puolesta monorepolle tyypillisen kansirakenteein Javascript-tai TypeScript-projektille vain yhdellä komennolla:



KUVA 3. Monorepon luominen käyttäen Turborepoa.

5.3 Tietokantapaketti



KUVA 4. Tietokantaskeeman määrittely Prisma-kirjastolla.

Tässä luvussa käsittelemme sovelluksen *database* pakettia, joka on tärkeä osa tyyppiturvallisuuden toteuttamista full-stack TypeScript-projekteissa. Tämä paketti sisältää tietokantaskeeman määrittelyn käyttäen Prisma-kirjastoa sekä Prisma Clientin exporttaamisen, jotta sen tarjoamia tyyppejä voidaan käyttää sekä frontend- että backend-puolella.

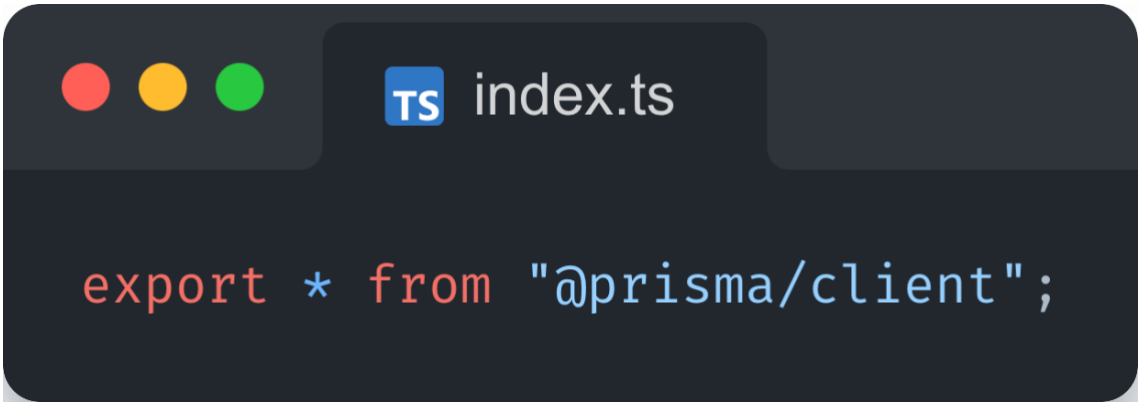
Prisma on moderni ja tyyppiturvallinen ORM (Object-Relational Mapping), joka helpottaa tietokantojen käyttöä TypeScript-sovelluksissa. Se tarjoaa vahvan tyyppijärjestelmän ja mahdollistaa tietokantaskeeman ja sovelluslogiikan integroimisen saumattomasti.

Tietokantaskeema kuvaa sovelluksen tietokannan rakenteen ja määrittelee taulujen väliset suhteet. Prisma-kirjastoa käyttäen voimme määritellä skeeman schema.prisma-tiedostossa, joka sisältyy *database*-pakettiin. Prisma tarjoaa oman DSL:n skeeman määrittelyyn, joka on sekä selkeä että helppolukuinen.

Esimerkissämme määrittelemme tietokantaan *Post*-nimisen taulun, ja sen eri kenttien tietotyypit. Prisma generoi määrittelyn pohjalta tietokantakyselyitä sekä tyyppimääritelmiä TypeScriptiä varten.

Kun tietokantaskeema on määritelty, voidaan Prisman komentorivityökalua käyttää Prisma Clientin generoimiseen. Prisma Client tarjoaa tyyppiturvallisen ja helppokäyttöisen API:n tietokantakyselyille. Generointi tapahtuu ajamalla prisma

generate komento komentorivillä ja se voidaan tehdä käytettäväksi tietokantapaketin ulkopuolella esimerkin tavalla.



KUVA 5. Prisma Clientin vienti tietokantapaketin ulkopuolelle.

Kun Prisma Client on viety tietokantapaketista, on se mahdollista tuoda sekä frontendiin että backendiin.

5.4 Tietokantapaketin käyttäminen backend-sovelluksessa

Tässä luvussa käsittelemme tietokantapaketin käyttöönottoa backend-sovelluksessa ja demonstroimme, kuinka voimme hyödyntää sen tarjoamia tyyppimääritelmiä ja Prisma Clientin API:a sovelluksen tietokantaoperaatioissa.

Ensimmäinen askel tietokantapaketin käyttöönotossa on asentaa se backend-sovellukseen. Käyttäessä monorepo-rakennetta ja pnpm workspaces -tekniikkaa, tietokantapaketti voidaan lisätä backend-sovelluksen riippuvuuksiin package.json-tiedostossa.

KUVA 6. Tietokantapaketin käyttöönotto package.json-tiedostossa.

Tämä määrittely kertoo, että database-paketti sijaitsee monorepon sisällä hakemistossa packages/database. Pnpm hoitaa riippuvuuksien linkityksen automaattisesti, joten kehittäjän ei tarvitse huolehtia siitä erikseen.

Kun tietokantapaketti on asennettu backend-sovellukseen, voidaan ottaa käyttöön sen tarjoamat tyypit ja Prisma Clientin rajapinnan. Tämä mahdollistaa tietokantaoperaatioiden suorittamisen tyypitettynä backend-sovelluksessa.

KUVA 7. Tietokantapaketin avulla tehty tyypitetty tietokantakysely.

5.5 Ajonaikainen tyyppitarkastus backend-sovelluksessa

Vaikka TypeScript tarjoaa tyyppiturvallisuuden kehitysaikana, se ei suojaa virheiltä, jotka johtuvat ajonaikaisista arvoista, kuten käyttäjän syötteistä. Tässä luvussa käsittelemme ajonaikaisen tyyppitarkastuksen tarpeellisuutta backend-

sovelluksessa ja esittelemme Zod-kirjaston, joka auttaa ratkaisemaan tämän ongelman.

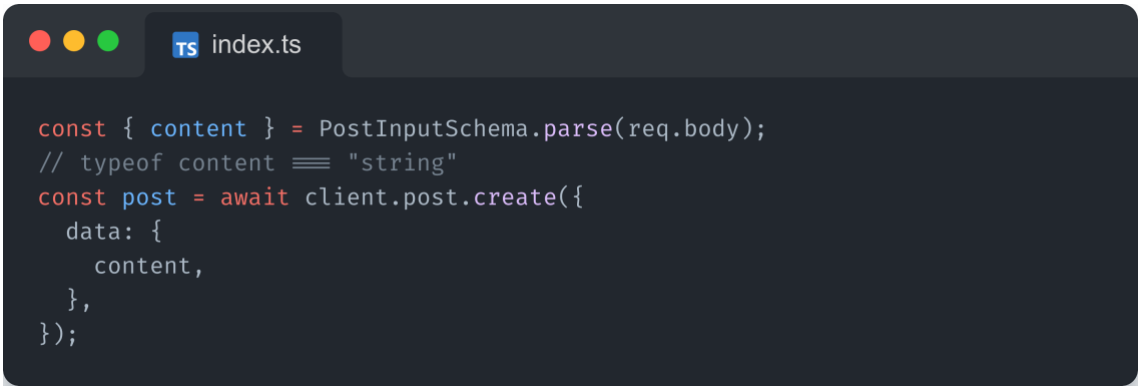
Kun sovellus vastaanottaa käyttäjän syötteitä tai kolmannen osapuolen API:lta tulevia tietoja, tietojen rakenne tai arvot eivät välttämättä täsmää odotettujen tyyppien kanssa. Tämä voi johtaa virheelliseen käyttäytymiseen tai jopa sovelluksen kaatumiseen.

Esimerkiksi, jos luodussa REST API:ssa odotetaan käyttäjän lähettävän syötteenä JSON-muotoisen käyttäjäobjektin, jolla on tietynlainen rakenne, emme voi olla varmoja siitä, että saamme oikeanlaisen objektin ilman ajonaikaista tarkastusta. Käyttäjä voi syöttää vääränlaisen arvon tai jättää jonkin kentän tyhjäksi, mikä voi aiheuttaa virheitä tai ongelmia sovelluksen toiminnassa.

Zod-kirjasto tarjoaa keinoja määritellä tyyppiskeemoja, joita voidaan käyttää ajonaikaiseen tyyppitarkastukseen. Zod-skeemat voivat tarkistaa, että annetut arvot ovat oikean tyyppisiä ja täyttävät määritellyt ehdot. Kun Zod on asennettu backend-sovellukseen, voidaan Zod-skeeman avulla määritellä käyttäjän syötteiden tai muiden ajonaikaisten arvojen tietotyypeille erilaisia vaatimuksia. Pelkästään oikean tietotyypin varmistamisen lisäksi Zod tarjoaa myös yksityiskohtaisempia ehtoja arvoille. Esimerkiksi merkkijonon tapauksessa on mahdollista määritellä vaatimukset sen pituudelle, ja jopa vaatia, että merkkijono sisältää vain hymiöitä.

KUVA 8. Skeeman määrittely Zod-kirjastolla.

Zodilla luodun skeeman käyttäminen sovelluksessa onnistuu kirjaston tarjoaman parse-metodin avulla. Sillä voidaan tarkastaa, vastaako metodille annettu syöte skeemamäärittelyä.



KUVA 9. Zod-skeeman käyttäminen sovelluksessa tyyppien validointiin.

Mikäli käyttäjän syöte ei vastaa skeemassa määriteltyä, Zod heittää virheen ja ohjelma ei pääse koodin suorituksessa eteenpäin.

KUVA 10. Zod-kirjasto heittää virheen, mikäli tyypit eivät vastaa skeemaa.

Zodin heittämien virheiden huomioonottaminen ja asianmukainen käsittely sovelluksessa on tärkeää, sillä näiden virheiden hallinta tässä kontekstissa on täysin kehittäjän vastuulla.

5.6 Tietokantapaketin käyttäminen frontend-sovelluksessa

Vaikka frontend-sovellus ei tee suoria tietokantakutsuja, tietokantapaketin tyypit ja rajapinnat ovat keskeisiä. Ne mahdollistavat API:sta saatavien tietojen käsittelyn oikein ja tyyppiturvallisesti. Tässä luvussa tarkastellaan tietokantapaketin integrointia frontend-sovellukseen sekä sen tyyppejä API-kutsuissa.

Monorepo-rakenteen ansiosta tietokantapaketin integrointi frontend-sovellukseen on suoraviivaista. Tämä paketti voidaan lisätä frontend-sovelluksen *package.json*-tiedostoon *dependencies*-osioon. Tällöin frontend-sovellus hyödyntää suoraan tietokantapaketin tarjoamia tyyppejä ja rajapintoja. Asennuksen jälkeen näitä tyyppejä ja rajapintoja voidaan soveltaa API-kutsuissa, parantaen sovelluksen tyyppiturvallisuutta.

Joskus kehittäessä saattaa tulla vastaan tilanteita, joissa sovelluksen tietokantapaketista saatu tyyppi onkin eri, kuin mitä frontendissä halutaan käyttää. On myös mahdollista, että jotakin tietokantapaketin tyyppiä ei haluta hyödyntää ollenkaan muualla. Tällaisiin tapauksiin TypeScript tarjoaa mahdollisuuden tyyppimuunnokselle käyttäen sen omia hyötyfunktioita. *Omit*-funktio mahdollistaa tiettyjen ominaisuuksien poistamisen tyyppimäärittelystä, luoden uuden tyypin, josta on jätetty pois tietyt avaimet.

Tietokantapaketista tuotuja tyyppejä voi käyttää frontendissä mm. React-sovelluksen tilan määrittelyssä. Kun tilalle on määritelty tyyppi, TypeScript osaa varoittaa esimerkiksi, jos yritämme käyttää jotakin tyypissä määrittelemätöntä kenttää. Tämä lähestymistapa auttaa pitämään frontend- ja backend-sovelluksien välisen tiedonkulun johdonmukaisena ja helposti ylläpidettävänä, sillä koska molemmat sovellukset käyttävät samoja tyyppimäärittelyjä, mm. tyyppimuutokset heijastuvat molempiin sovelluksiin.

posts.tsx

```tsx
export default function Posts() {
  const [posts, setPosts] = useState<Post[]>([]);
  const [inputValue, setInputValue] = useState<string>("");

  useEffect(() => {
    fetch("http://localhost:3333/posts")
      .then((res) => res.json())
      .then(setPosts);
  }, []);

  return posts.map((post) => <Post key={post.id} text={post.content} />);
}
```

KUVA 11. Tietokantapaketin tyyppien käyttäminen React-komponentissa.

Mikäli kehittäjä tekee tietokantaskeemaan jonkin muutoksen, esimerkiksi vaihtaa jonkin taulun kentän nimeä, TypeScript varoittaa tästä kaikkialla, missä vanhaa kentän nimeä käytetään.

KUVA 12. Tyyppivirhe React-sovelluksessa tyyppimuutoksen jälkeen.

6 POHDINTA

Tietokoneohjelmien vahva tyypitys ja sitä kautta saavutettava tyyppiturvallisuus on yksi ohjelmistokehityksen keskeisistä aspekteista, ja sen rooli on viime vuosina korostunut erityisesti moderneissa ohjelmointikielissä kuten TypeScriptissä. Tämän opinnäytetyön tavoitteena oli paitsi valottaa tyyppiturvallisuuden merkitystä, myös esitellä konkreettisia tapoja sen toteuttamiseen FullStack TypeScript -sovelluksissa.

FullStack TypeScript -kehityksessä tyyppiturvallisuuden toteutuminen sekä frontendissä että backendissä on haastavaa, koska järjestelmän eri osat kommunikoivat tiiviisti keskenään, ja niiden koodit ovat usein erillään toisistaan. Esimerkkiprojektissa käytössä ollut monorepo-strategia kuitenkin paljasti, että se voi tarjota tehokkaita keinoja tyyppiturvallisuuden varmistamiseksi koko projektin laajuisesti. Toisaalta monorepon käyttöönottoon liittyy omat haasteensa; se voi viedä paljon aikaa, ja etenkin jos monorepo halutaan ottaa käyttöön vanhassa projektissa, joudutaan se usein tekemään hiljalleen muun kehityksen ohessa.

Opinnäytetyön aikana havaittiin, että eräs erittäin keskeinen hyöty web-sovelluksen vahvassa tyypityksessä löytyy kehittäjäkokemuksen parantumisessa. Useimmat tekstieditorit pystyvät tarjoamaan kehittäjälle paljon lisäominaisuuksia, jotka helpottavat ja nopeuttavat kehitystyötä. Kun TypeScriptin kielipalvelimen tarjoamia ominaisuuksia, kuten koodin automaattinen täydennys tai tyyppimäärittelyiden näkeminen suoraan editorissa, on hetken käyttänyt, huomaa kehittämisen nopeuden laskevan huomattavasti, jos näitä ominaisuuksia ei enää olekaan saatavilla. Tyyppiturvallisuuden saavuttaminen ei kuitenkaan ole ratkaisu vain täysin teknisiin haasteisiin. Se vaikuttaa myös kehittäjien työskentelytapoihin, yhteistyöhön ja koko tiimin dynamiikkaan, kun kaikki ovat tietoisia, miten ohjelman tulisi toimia.

On tärkeää myös korostaa, että tyyppiturvallisuuden merkitys ei rajoitu pelkästään ohjelmistokehitykseen. Se vaikuttaa koko sovelluksen elinkaareen, mukaan lukien testaus, ylläpito ja jatkokehitys. Tulevaisuudessa, kun ohjelmistoprojektien kompleksisuus ja digitaalisuus ylipäätään jatkaa kasvuaan ja teknologiat kehittyvät, tyyppiturvallisuuden rooli korostunee entisestään. Tätä

nousevaa trendiä on ollut havaittavissa jo viime vuosina, kun staattisesti tyypitetyt kielet kuten Go ja Rust ovat kasvattaneet suosiotaan. Nykyään myös harvemmin näkee, että uutta projektia aloitetaan enää JavaScriptiä käyttäen. Lähes kaikki ovat sen sijaan siirtyneet käyttämään TypeScriptiä, oikeutetusti.

LÄHTEET

Cardelli, L. 2004. Type Systems (Microsoft Research).

Ecma International. 2023. ECMAScript 2023 Language Specification. Viitattu 23.4.2023. https://262.ecma-international.org

Jansen, R., Vane, V. & de Wolff, I. 2016. TypeScript: Modern JavaScript Development. Birginham: Packt.

Leveson, N, 2010. Medical Devices: The Therac-25, 1–10.
MongoDB, Inc. n.d. Full Stack Development Explained. Viitattu 23.9.2023. https://www.mongodb.com/languages/full-stack-development

npm Inc. n.d. npm. Verkkosivu. Viitattu 20.9.2023. https://www.npmjs.com

Oracle. 2015. Dynamic typing vs static typing. Verkkosivu. Viitattu 22.4.2023. https://docs.oracle.com/cd/E57471_01/bigData.100/extensions_bdd/src/cext_transform_typing.html

pnpm. n.d. Workspace | pnpm. Verkkosivu. Viitattu 24.9.2023. https://pnpm.io/workspaces

Pocock, M. 2023. TS Reset Improves TypeScript's Built-In Typings. Total TypeScript 17.8.2023. Viitattu 22.9.2023. https://www.totaltypescript.com/ts-reset-article

Prisma Data, Inc. 2022. Top 11 Node.js ORMs, query builders & database libraries in 2022. Viitattu 24.9.2023. https://www.prisma.io/dataguide/database-tools/top-nodejs-orms-query-builders-and-database-libraries

Prisma Data, Inc. 2023. Building fullstack applications with Prisma. Verkkosivu. Viitattu 24.9.2023. https://www.prisma.io/docs/concepts/overview/prisma-in-your-stack/fullstack

Stack Overflow. 2020–2023. Stack Overflow Developer Survey. Viitattu 20.9.2023. https://survey.stackoverflow.co

Turun Sanomat. 2005. Ensimmäinen ohjelmistovirhe löytyi 60 vuotta sitten. Turun Sanomat 22.11.2005. Viitattu 20.9.2023. https://www.ts.fi/uutiset/1074083485

TypeScript. n.d. TypeScript Handbook. Verkkosivu. Viitattu 23.4.2023. https://www.typescriptlang.org/docs/handbook/

TypeScript. n.d. TypeScript: JavaScript With Syntax For Types. Verkkosivu. Viitattu 22.4.2023. https://www.typescriptlang.org

Zod. n.d., Zod Documentation.Verkkosivu. Viitattu 21.9.2023. https://zod.dev