

Ruby on Rails -sovelluskehys; case: Translator

Kristian Wahlroos - 014417003

Helsinki 22.1.2016

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Sisältö

1	Johdanto	1
1.1	Translator -projekti	1
1.2	Ruby on Rails -sovelluskehys	3
1.3	Arkkitehtuuriset tyylit	6
2	Yleisarkkitehtuuri ja keskeisimmät variaatiopisteet	8
2.1	Ohjelmistotason suunnittelumallit	11
2.2	Kehyksen erikoistaminen sovelluskohtaisesti	11
2.3	Sekvenssikaavio	12
3	Kehyksen ja sovelluksen arviointi	14
3.1	Hyvät ja huonot puolet	14
3.1.1	Hyvät puolet	14
3.1.2	Huonot puolet	15
3.2	Arviointi	16
3.2.1	Muodostettujen skenaarioiden analysointi	17
3.2.2	Skenaarioista johdetut päätelmät	18
	Lähteet	19

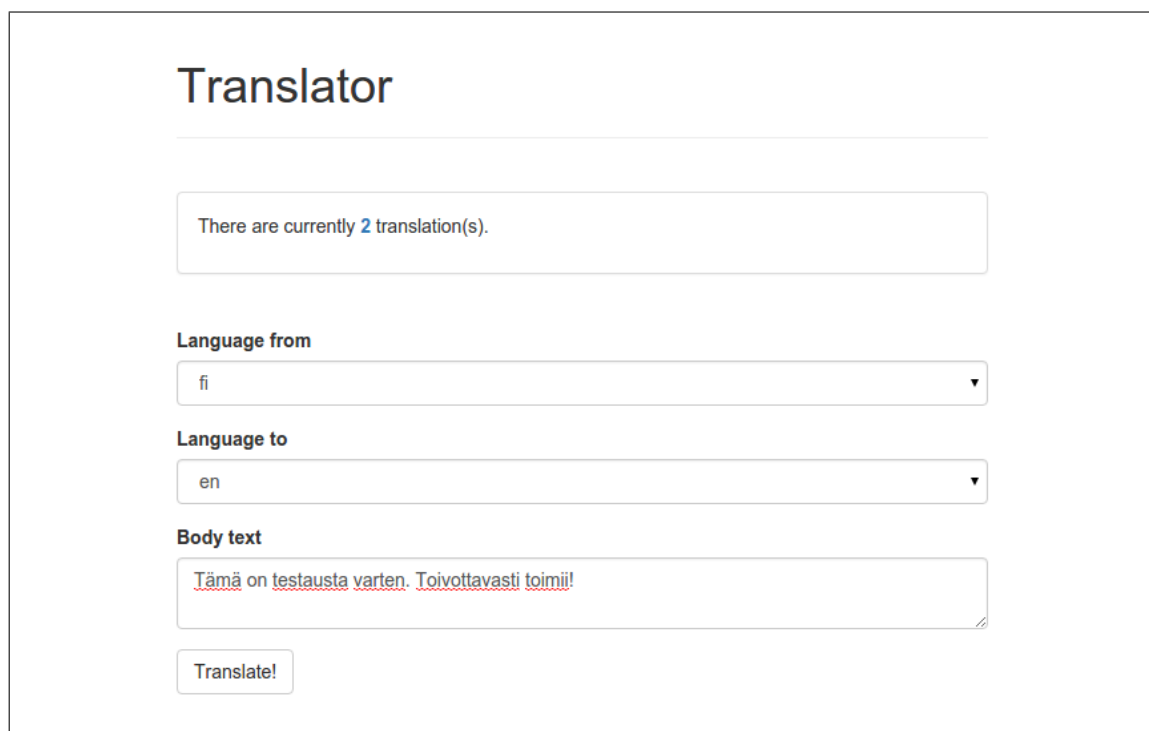
1 Johdanto

Tämän harjoitustyön tarkoituksena on tarkastella Ruby on Rails -web sovelluskehystä ja varsinkin sen arkkitehtuuria tekemäni esimerkksiovelluksen kautta. Tekemäni esimerkksiovellus on yksinkertainen Internetissä oleva käännössivusto, jonka avulla käyttäjä pystyy kääntämään kolmen eri kielen välillä; suomen, englannin ja ruotsin. Itse sivusto on erittäin yksinkertainen eikä toiminnallisuutta ole hirveästi kääntämisen lisäksi. Raportin keskittyminen tapahtuukin sovelluksen pinnan alle kuten arkkitehtuuriin ratkaisuihin joita Ruby on Rail-kehys tuo mukanaan.

1.1 Translator -projekti

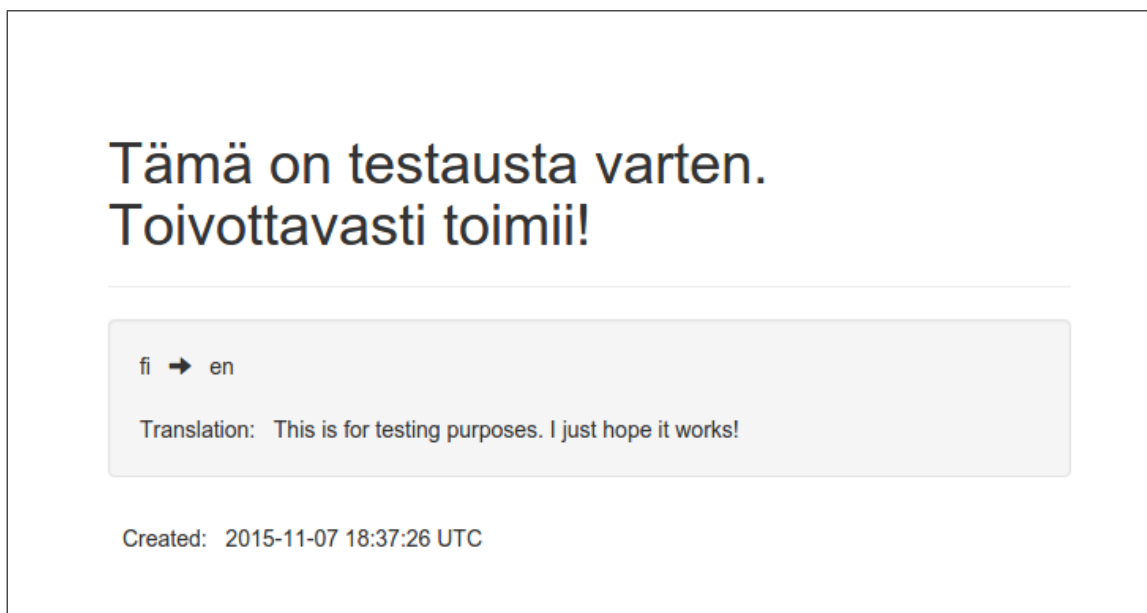
Sovellukseneeni viittaa tästä alkaen nimellä *Translator*.

Translatorin pääidea on siis nimensä mukaisesti kääntää annettu teksti kieleltä toiselle. Käännöksen logiikka hoidetaan ulkopuolista API:a käyttäen [YAPI], joka hoitaa käytännössä kääntämisen sovelluslogiikan. Seuraavassa esitettävä kuvasarja esittää kääntämisen käyttäjän näkökulmasta:



The screenshot shows a web application titled "Translator". Below the title, there is a status message: "There are currently 2 translation(s)". Underneath, there are two dropdown menus for language selection. The first is labeled "Language from" and has "fi" selected. The second is labeled "Language to" and has "en" selected. Below these is a text input field labeled "Body text" containing the text "Tämä on testausta varten. Toivottavasti toimii!". At the bottom of the form is a button labeled "Translate!".

Kuva 1: Käyttäjä haluaa kääntää lauseen suomesta englantiin. Osoite: /



Kuva 2: Uudelleenohjataan selain näyttämään tulos. Osoite: /translations/:id

From	Body text	To	Translation	
fi	hei nimeni on Kaapo	en	hey, my name is Kaapo	View
fi	Hei maailma!	en	Hello world!	View
fi	Tämä on testausta varten. Toivottavasti toimii!	en	This is for testing purposes. I just hope it works!	View

Kuva 3: Käyttäjä on navigoinut itsensä kaikkien käännösten sivulle klikkaamalla linkkiä pääsivulla (linkki on tallennettujen käännösten määrä). Osoite: /translations.

Edellinen kuvasarja eteni siis käyttäjän selaimessa seuraavasti:

Pääsivu (/) → tallennettu käännös (/translations/:id, jossa :id korvautuu tietokantaan tallennetun tietueen id:llä) → pääsivu (/) → kaikki käännökset (/translations).

1.2 Ruby on Rails -sovelluskehys

Ruby on Rails on itsessään sovelluskehys, joka on luotu vuonna 2003 David Heinemeier Hansson toimesta [ROR]. Sen keskeisimpiä ominaisuuksia ovat MVC-malli, REST-rajapinnat, Convention over Configuration- ja Don't Repeat Yourself-periaate. Nämä kaikki tulevat esille käytännössä kaikissa Ruby on Rails -kehyksellä tuotetuissa projekteissa ja moni tunteeikin kehyksen juuri näistä edellä mainituista laatuviuista, joita kehyksen käyttö melko automaattisesti tuo mukanaan.

Yleisimpiä arkkitehtuurisia tyylejä ei sovelluskehyksestä suoraan löydy, koska kehys on sidottu lujasti MVC-patterniin. Kuitenkin asiakas-palvelin -tyyli on melko selkeästi nähtävissä Ruby on Railsissä.

Model-View-Controller -patterni Ruby on Rails:n Modelina toimii ActiveRecordin avulla käsiteltävät tietokantaobjektit. ActiveRecord huolehtii käytännössä koko toteutettavan järjestelmän logiikasta ja abstrahoi vahvan rajapinnan avulla konkreettista tietokantaa niin, että kehittäjän on esimerkiksi helppo vaihtaa tietokanta toiseen versioon sekä luoda uusia tietokantaobjekteja.

Kehyksen ActiveRecord itsessään toteuttaa Active Record-patternin [ARP], joka määrittelee, että miten luokat ja tietokanta kuvautuu toisilleen. Railsissa tämä kuvautuminen on tehty niin, että tietokannan tietue on aina luokka ja taulut luokan kenttiä. Luokkien metodeilla usein muutetaan vain ja ainoastaan tietokantataulun rivejä, jolloin olion sisäinen tila muuttuu. Luomassani projektissa uuden käännöksen ja sen tallentaminen tietokantaan on hyvin yksinkertaista tämän vuoksi, eikä kehittäjän tarvitse tietää, että taustalla pyörii SQLite tietokantana:

```

1 t = Translation.new
  ***
3 kenttien alustus
  ***
5 t.save

```

ActiveRecordin avulla myös suorat tietokantakyselyt on abstrahoitu pois. Kyselyt toimivat aina luokan nimen kautta, jolloin esimerkiksi omassa sovelluksessani kaikki suomesta käännetyt käännökset löytyvät seuraavasti:

```
1 Translation.all.where language_from: "fi"
```

Viewin vastuuta Ruby on Railsissä hoitaa ActionView, joka on näkymä tietokannan datalle. Kaikessa yksinkertaisuudessaan siis näytettävä HTML-sivu käyttäjälle. Se ladataan kontrollerin toimesta oikeasta kansioista oikealla hetkellä ja käytännössä se onkin HTML-tiedosto, joka sisältää upotettua Ruby-koodia. Tämä Ruby-koodi on suurimmaksi osaksi toiminnallisuutta näyttää vastaavan mallin tietoja HTML-muodossa. Esimerkiksi sovellukseni kaikki tehdyt käännökset näyttävä sivu on suurimmaksi osaksi upotettua Ruby-koodia, joka iteroi tietokannan kaikki käännökset ja hakee niiden tietokantataulut taulukkoon näytettäväksi dataksi.

Controlleria kutsutaan ActionControlleriksi Ruby on Railsissa ja se vastaa koko järjestelmään kohdistuvien pyyntöjen reitittämisestä sekä yleisestä datan välityksestä. Omassa järjestelmässäni TranslationsController vastaa kaikesta käännöksiin liittyvästä toiminnallisuudesta, kuten esimerkiksi aloitussivun näyttämisen logiikasta. Railsissa konventiot tulevatkin vahvasti esille, koska Translation-luokasta vastaavan kontrollerin on oltava samanniminen kuin luokkakin, mutta monikossa (englannin-s-päätteellä). Tähän kontrolleriin on myös kirjoitettu kaikki mahdolliset toiminnot, joita mallille on mahdollista tehdä ja näiden avulla rakennetaan myös toimintoja vastaavat näkymät.

REST-rajapinta Koko Ruby on Railsin käyttöönotto ja sen konventioiden noudattaminen tekee toteutettavasta järjestelmästä melkein automaattisesti REST-rajapintaa noudattavan järjestelmän, koska useat konventiot ovat pakollisia käyttää ja vaikka niiden kiertäminen on teknisesti mahdollista, on usein vain helpompi alistua valmiiksi määriteltyihin konventioihin. Esimerkiksi luodessani Translation-mallia, oli minun pakko luoda TranslationsController-kontrolleri, mutta ennen niiden linkittämistä on tiedostoon /config/routes.rb kerrottava, että mikä osoite linkittyy mihinkin kontrolleriin. Omassa projektissani olen lisännyt seuraavan rivin kyseiseen tiedostoon:

```
1 resources :translations, only: [:show, :new, :create]
```

Tämän avulla olen automaattisesti saanut käyttöön seuraavat osoitteet, jotka nou-

dattavat REST-tyyliä:

1	translations	POST	/translations (::format)	translations#create
	new_translation	GET	/translations/new (::format)	translations#new
3	translation	GET	/translations/:id (::format)	translations#show

Convention over Configuration Konventiot ovat tärkeässä roolissa Ruby on Rails -sovelluskehityksessä. Ne suoraviivaistavat koko toteutettavan järjestelmän arkkitehtuuria, vähentävät ohjelmakoodin määrää, vähentävät toistoa ja tuovat yhtenäisen toimintatavan, jonka avulla jokainen Ruby on Rails:llä tuotettu järjestelmä on hyvin samankaltainen muiden Rails-sovelluskehityksellä tuotettujen kanssa. Esimerkiksi nimeämiseen liittyvät konventiot ovat erittäin näkyvässä roolissa ja osittain jopa pakollisiaakin [RARN]. Nimeämiset näkyvät projektissani esimerkiksi, että projektini tietokantataulussani 'translations' sijaitsevat kaikki Translation-mallin ilmentymät, kontrolleri nimeltä 'TranslationsController' huolehtii näiden mallien päivittämisestä sekä oikeiden näkymien näyttämisestä.

Nimeämiskonvention avulla sovelluskehitys osaa automaattisesti ohjata sovelluksen toimintaa haluttuun suuntaan, jolloin tarve monimutkaisille konfiguraatio-tiedoistoille vähenee. Ruby on Rails siis pinnan alla automaattisesti päättelee nimien perusteella, että kenen vastuulle tiettyyn pyyntöön vastaamisen kuuluukaan.

Don't Repeat Yourself DRY-periaate näkyy vahvasti muun muassa Gemfile:n, routes.rb:n sekä tietokantamigraatioiden kautta. Gemfile-tiedostossa sijaitsee sovelluksen kaikki tarvitsemat Gem:it eli ulkoiset kirjastot, routes.rb-tiedosto taas sisältää kaikki määrittelyt pyyntöjen ohjaukseen oikealle kontrollerille ja tietokantamigraatioille on myös oma paikkansa 'db'-kansion alla. Migraatioiden avulla tehdään muutoksia tietokantaan ja sieltä nähdään kaikki aiemmin tehdyt muutokset ja niiden peruutukset ('rollbackit'). Nämä tiedostot vähentävät turhaa määrittelyä ulkopuolisiin luokkiin, jolloin konfiguraatiot sijaitsevat aina yhdessä paikassa.

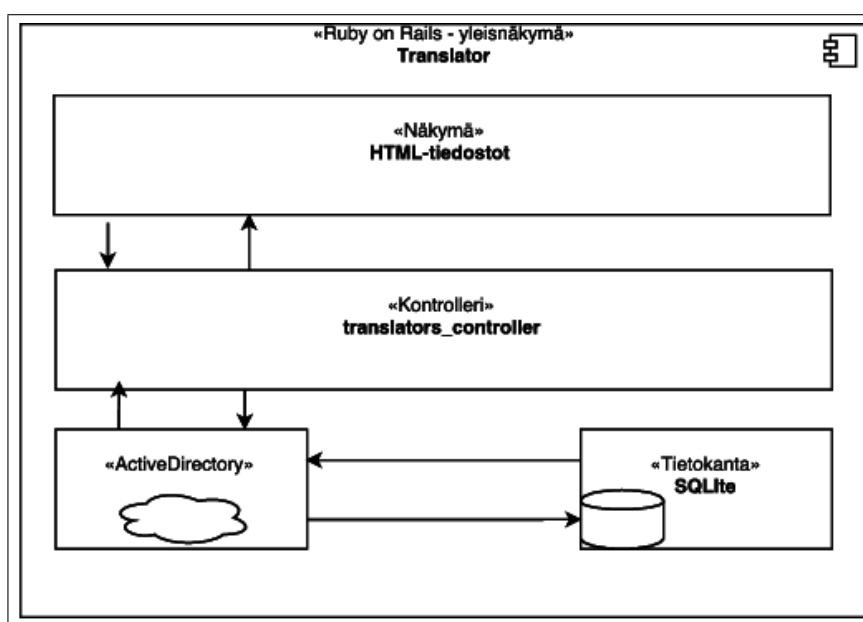
Toisaalta jo aiemmin mainittu ActiveRecord vähentää myös toistoa, koska kehittäjien ei tarvitse kirjoittaa erikseen ylös tietokantatauluja mallien määrittelytiedostoihin, vaan Ruby on Rails hoitaa kaiken tietokantalogiikan nimeämiskonventioiden avulla. Konventiot yleisestikin vähentävät koodin määrää, joka taas mahdollistaa monimutkaisten asioiden tekemisen vain muutamalla rivillä ohjelmakoodia.

1.3 Arkkitehtuuriset tyylit

Ruby on Railsistä ei suoraan löydy juuri niinkään yhtä tiettyä tyyliä, jota kehys noudattaisi, mutta kehystä voidaan tarkastella 3-taso -ja asiakas-palvelin -mallin kautta.

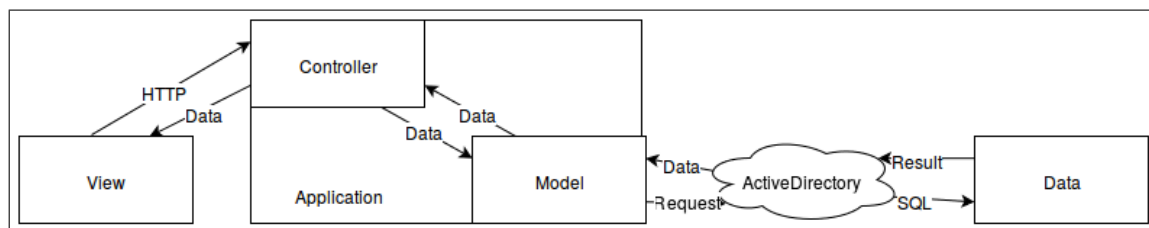
3-taso -malli Vastuut ovat kolmitasoisena tarkasteltuna jaettu niin, että näkymä-tasosta huolehtivat HTML-tiedostot, sovellustasosta huolehtivat mallit sekä kontrollerit. Kontrollerit saavat pyyntöjä UI:lta (HTML/HTTP) ja niissä tapahtuu mallien muokkaus. Datataso Railsissä koostuu ActiveDirectorystä, tietokannasta sekä varsinakin malleista. Mallit siis kuuluvat periaatteessa sovellustason sekä datatason välimaastoon, koska toisaalta kaikki sovelluslogiikka sijaitsee niissä, mutta toisaalta ne toimivat abstraktiona tietokannan tauluille ja koko tietokannan muokkaus tapahtuu suoraan mallien kautta.

Itse kerroksia ei pysty mitenkään ohittamaan, jolloin datan tallennus suoraan näkymästä on esimerkiksi käytännössä mahdotonta. Seuraavassa kuvassa on havainnollistettu tekemäni järjestelmää 3-taso -mallin avulla, josta nähdään tämä kyseinen rajoite.



Kuva 4: Yksinkertaistettu arkkitehtuuri Translator-sovelluksesta 3-tasoisena tarkasteltaessa. Datan virtaus kuvattu nuolilla. Alin taso huolehtii tiedon tallennuksesta, keskimäinen tiedon käsittelystä ja ylin tiedon näyttämisestä.

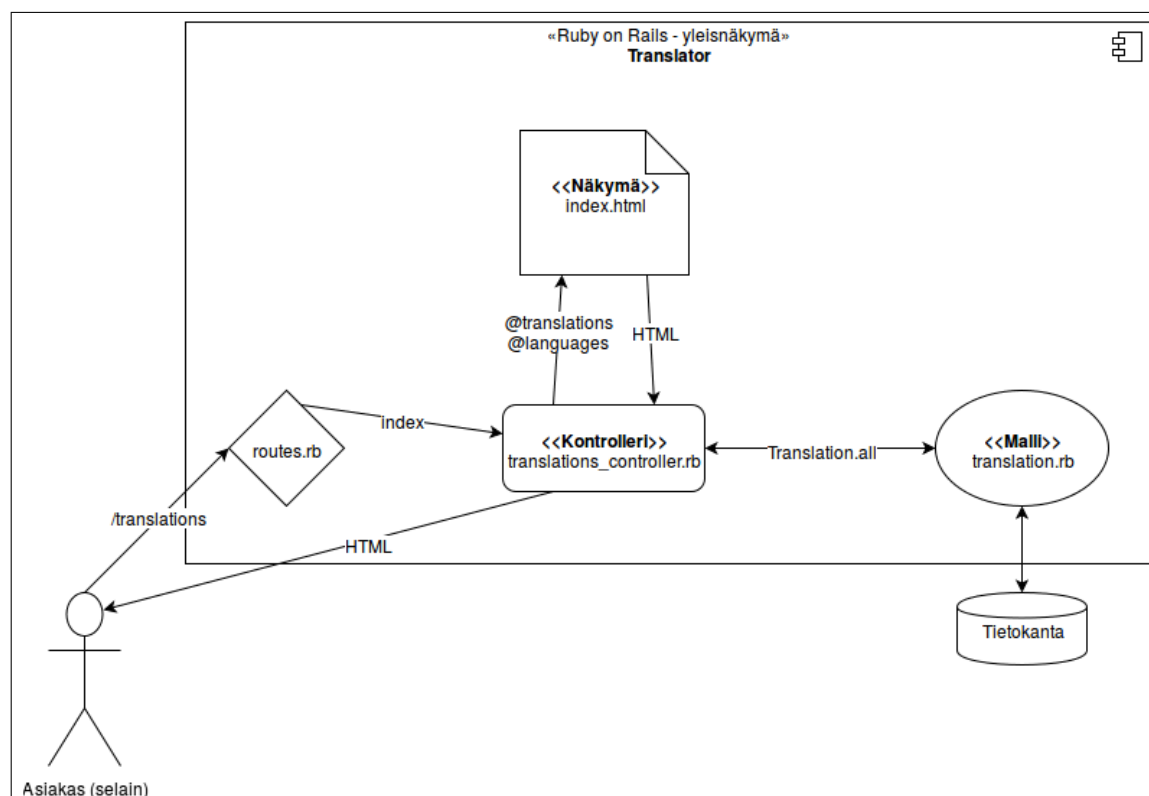
Asiakas-palvelin Kommunikointi kaikkien komponenttien välillä tapahtuu asiakas-palvelin- tyylin perusteella. Kontrollerit odottavat pyyntöjä näkymiltä ja tarjoavat tiedon näkymille mallien kautta. Mallit taas ovat suora linkki tietokantaan tehden muokkauksia sinne mallia vastaavaan tietokantatauluun. Mallit hakevat myös tietoa kontrollerien käyttöön ja täten tarjoavat kontrollereille muokkaus/haku-rajapinnan tietokantaan. Tämä kyseinen kommunikaatio tietokantatauluun tapahtuu aina ActiveDirectoryn kautta, joka muuttaa pyynnöt kyseisen tietokannan osaamalle kielelle (usein SQL).



Kuva 5: Havainnekuva kommunikaatiosta.

2 Yleisarkkitehtuuri ja keskeisimmät variaatiopisteet

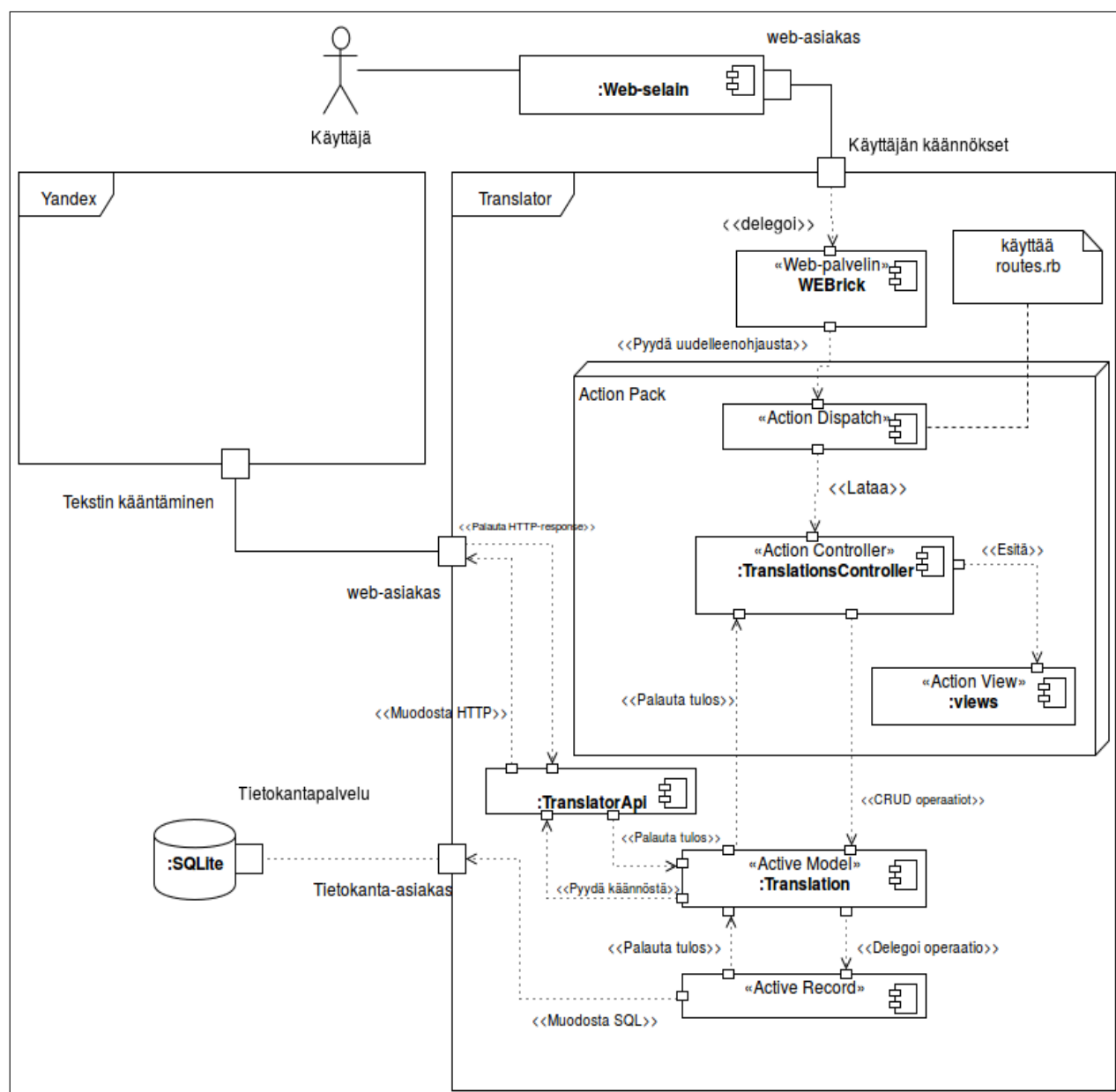
Ruby on Rails toteuttaa yleiseltä arkkitehtuuriltaan erittäin vahvasti jo aiemmin mainittua Model View Controller -tyyliä ja todella moni kehyksellä tehdyistä sovelluksista onkin arkkitehtuuriltaan MVC-mallinen. MVC-mallisena tarkasteltaessa Ruby on Rails -sovelluskehys näyttää seuraavanlaiselta:



Kuva 6: Translator-sovellus MVC-mallin silmin. Kuvassa käyttäjä avaa selaimensa `/translations`-osoitteeseen, joka reititetään oikealle kontrollerin metodille (`index`) `routes.rb`-tiedoston avulla. Kontrolleri hakee kaikki `Translation`-mallin ilmentymät mallin avulla tietokannasta. `@translations` sekä `@languages` muuttujat välitetään tämän jälkeen näkymälle, joka luo halutun `HTML`-tiedoston. Lopuksi kontrolleri palauttaa tämän dokumentin käyttäjän selaimelle.

Sisärakennemalli Seuraavassa tarkastellaan Translator-järjestelmää komponentteittain, josta erottuu kaksi isoa järjestelmää. Toinen on Translator-sovellukseni ja toinen Yandex-API, joka huolehtii kääntämisen logiikasta. Näiden järjestelmien väli-

sestä kommunikaatiosta huolehtii TranslatorApi-komponentti, joka muodostaa tarvittavat HTTP-pyyntöt ja palauttaa niiden tuloksen Translation-mallille sellaisessa muodossa, että se pystytään välittämään järkevässä muodossa eteenpäin.



Kuva 7: Yhtenäinen viiva tarkoittaa web-pyyntöä ja katkonainen tietokantayhteyttä. Muut viivat komponenttien välisiä funktiokutsuja, joiden sisältö «»-välissä.

Translator-sovelluksesta voidaan erotella seuraavat Ruby on Rails -kehiksen kannalta pakolliset komponentit, jotka mahdollistavat pohjatoiminnallisuuden koko järjestelmälle. Osa näistä ei näy kehittäjälle välttämättä ollenkaan, jolloin kehittäjä voi

keskittyä vain oman järjestelmänsä kannalta relevantteihin asioihin.

Web-palvelin Oletuksena WEBrick Ruby on Rails -kehyksessä [WEBR]. Tarjoaa yksinkertaisen HTTP-palvelimen toiminnallisuuden. Voidaan helposti korvata esimerkiksi Apache tai Puma-palvelimella [RDWS], sekä usealla muulla sovelluskehysten tukemalla.

Action Pack Tarjoaa kontrolleri- ja näkömäkerroksen Model-View-Controller-patternille. Tässä sijaitsevat komponentit käsittelevät selaimen pyynnöt ja reitittävät pyynnöt oikeille komponenteille jatkokäsittelyä varten, mutta myös vastaavat näkymän toteuttamisesta. Action Pack jakautuu kolmeen eri rajapintaan: Action Dispatch, Action Controller ja Action View.

Action Dispatch Hoitaa selaimelta tulevien HTTP-pyyntöjen reitittämisen oikealle kontrollerille sekä niiden jäsentelyn. Suurin osa toiminnallisuudesta on määritelty routes.rb-konfiguraatiotiedostoon.

Action Controller Määrittelee kontrollereiden perustoiminnallisuuden, jota voidaan periyttämällä erikoistaa. Toiminnallisuus koostuu näkymiä ja malleja varten tehdyistä operaatioista. Esimerkiksi datan saaminen näkymälle asti sekä uudelleenohjauksen toiminnallisuuden.

Action View Tulee kutsutuksi Action Controllerin kautta. Muodostaa halutun näkymän www-sivustosta käyttäjälle käyttäen hyödykseen erilaisia apukeinoja (esimerkiksi lomakkeiden ja linkkien muodostamiseen), templaatteja sekä hyvin usein RHTML-muotoa. RHTML on muodostettua HTML-koodia, jossa on upotettua ruby-koodia seassa, joka sitten korvautuu täysin HTML-muotoon juuri ennen lopullista esittämistä [RHTML].

Active Model Määrittelee rajapinnan edellä mainitun Action Pack-kokonaisuuden sekä Active Recordin välillä. Kaikki mallit, jotka haluavat toimia tietokannan kautta toteuttavat tämän rajapinnan.

Active Record Huolehtii datan muokkauksesta objektien kautta ja tarjoaa CRUD-rajapinnan (Create Read Update Delete) tietokantaan. Jokainen operaatio tapah-

tuu aina luokan kautta, jolle löytyy vastaavuus tietokannassa. ActiveRecord luottaa paljon nimeämiskonventioihin muualla sovelluksessa, jolloin erinäisiä konfiguraatio-tiedostoja ei tarvita.

2.1 Ohjelmistotason suunnittelumallit

Rakentamassani järjestelmässä ei ole tarkoituksella käytetty mitään tiettyä ohjelmistotason suunnittelumallia, mutta itse Rails-kehyskään ei sisäisesti toteuta raportin alussa mainittujen lisäksi muita, koska iso osa joustavuudesta tulee jo itse Ruby kielestä ja sen ominaisuuksista. Toisaalta kuitenkin ActiveRecord toteuttaa Object-relational mapping -mallia piilottaessaan tietokantakyselyt luokkien taakse, mutta tämän lisäksi ei muita ohjelmistotason suunnittelumalleja löydy.

2.2 Kehyksen erikoistaminen sovelluskohtaisesti

On huomattavaa, että esimerkiksi tietokanta ei sisälly järjestelmän sisälle, vaan ulkopuolelle ja on näin helposti vaihdettavissa. Tämä on toteutettu aiemmin mainitun ActiveRecordin avulla. Näkymien vaihdettavuus on erittäin helppoa, koska Ruby on Rails-konventioiden mukaan vain nimeämisellä (ja kansiorakenteella) on väliä, joten kunhan tiedostot sijaitsevat oikeissa kansioissa oikeilla nimillä, niin esimerkiksi näkymät saadaan vaihdettua muuttamalla vain tiedoston nimeä.

Toisaalta laajemmin ajateltuna ainoaksi yhteiseksi osaksi variaatiopisteiden kannalta jää malli, koska se on ainut osa, joka toteuttaa itse business-logiikan ja toimii abstraktiona tosimaailman käännöksen käsitteelle. Järjestelmässäni Translation-malli on hyvin sidottu, koska haluan sovelluksen nimenomaan olevan käännös-toimintoa tukeva sivusto, joka pystyy tallentamaan aiemmin tehtyjä käännöksiä. Tämän vuoksi tallenuslogiikalla, näkymillä, eikä ohjainkomponenteille ole kovinkaan tärkeää roolia järjestelmässä ja tästä syystä ne voidaan laskea vaihtuviksi osiksi variaation kannalta Ruby on Rails-kehyksessä.

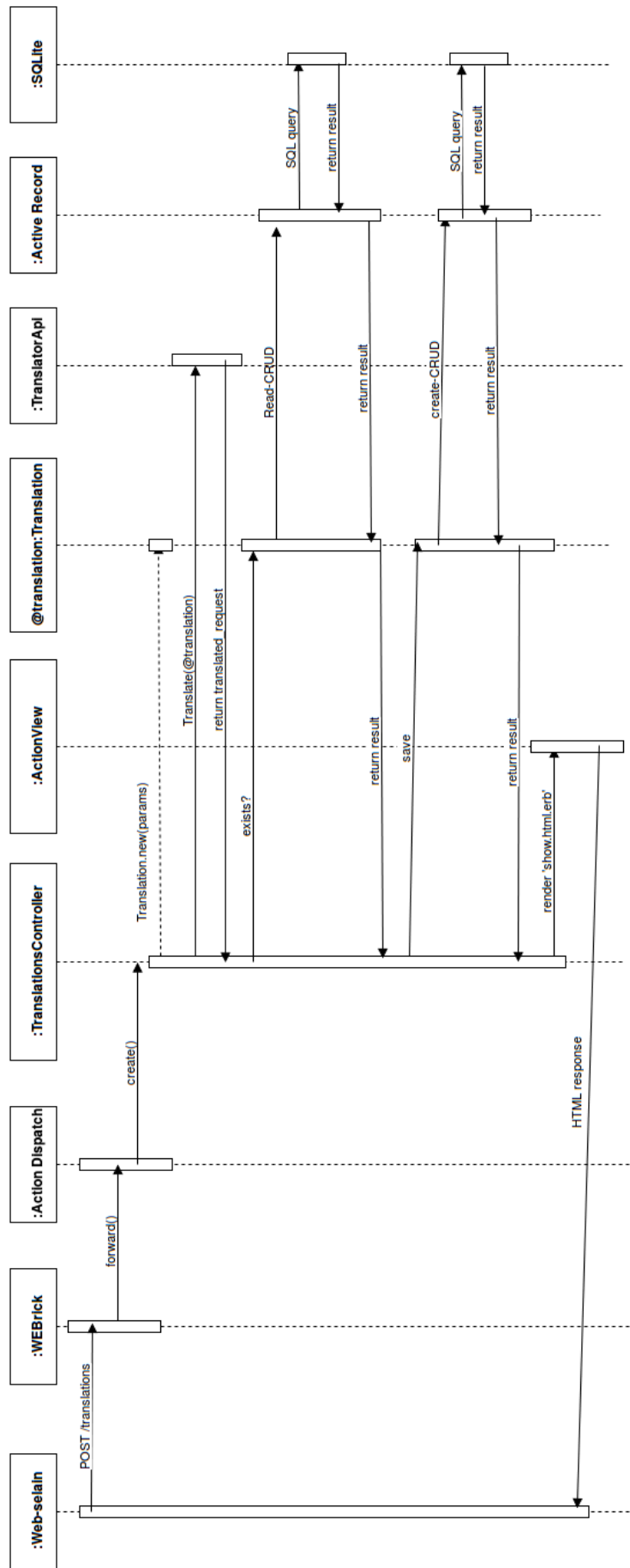
Itse kehyksen erikoistaminen tapahtuu pääosin periyttämällä valmiita rajapintoja. Esimerkiksi mallit perivät Active Model-rajapinnan, jotta niihin saadaan oikea toiminnallisuus, mutta jos kehittäjä haluaa täysin itsenäisiä luokkia, niin silloin luokkien ei tarvitse periä mitään. Tällöin niistä tosin katoaa valmista toiminnallisuutta, mutta myös rajoitteet katoavat. Katoavana rajoitteena esimerkiksi nimeäminen, joka vapautuu täysin kehittäjän päätettäväksi.

Kuvassa 7 olen merkannut kuvaan järjestelmän sisälle tummennettuna ne komponentit, joihin olen tehnyt itse muutoksia. Kuitenkin `TranslationsController`, `views` ja `Translation` ovat perittyjä jostain tai toteuttavat jonkun rajapinnan, jolloin niiden toiminnallisuus on suhteellisen rajoitettu ja ennalta määriteltyä. Ainoastaan `TranslatorApi` on täysin itsenäinen oma staattinen luokkansa, jota `Translation` käyttää hyväkseen muodostaessaan kyselyitä ulkopuoliseen Yandex-APIin.

2.3 Sekvenssikaavio

Seuraavalla sivulla kuvataan sekvenssikaavion avulla miten uuden käännöksen tekeminen tapahtuu järjestelmässä ja miten se tallentuu tietokantaan. Kuvassa käyttäjä on navigoinut itsensä sivulle etusivulle (`/`-osoitteeseen). Hän kirjoittaa lomakkeeseen haluamansa tekstin, valitsee kielen ja painaa `Translate!`-nappia (Kuva 1). Sekvenssikaavio alkaa tästä tapahtumasta, mutta siinä on abstrahoitu joitain osia pois (funktiokutsuja) pääosin sen takia, että kuvaa olisi helpompi tulkita.

On huomattavaa, että juuri kontrolleri luo instanssin `Translation`-mallista käyttäjän syöttämällä parametreilla ja hoitaa funktiokutsun `TranslatorApi`-luokalle parametrinaan juuri luotu instanssi. Tässä tulee esille kontrollerin toiminta juuri tietovuon ohjaajana ja hallitsina, kun taas itse kääntäminen tapahtuu mallin kautta (`Translation`-mallilla julkinen `translate`-metodi).



Kuva 8: Sekvenssikaavio käännöksen tekemisestä ja tallentamisesta.

3 Kehyksen ja sovelluksen arviointi

Sovelluskehiksenä Ruby on Rails on erittäin tarkka sen omista käytännöistä, kun sitä verrataan esimerkiksi toiseen erittäin suosittuun backend-sovelluskehikseen NodeJS:ään. NodeJS tarjoaa samalla tavalla kehiksen tuottaa web-palveluita, mutta se on rakenteellisesti paljon kevyempi ja vähemmän rajoittava. Toisaalta Ruby on Rails sopii hyvin aloitteleville ohjelmoijille sekä pienehköille start-up -henkisille yrityksille juuri sen käytänteiden vuoksi. Niitä noudattamalla pystytään todella nopeasti kehittämään palveluita asiakkaille samalla kaavalla.

Kuitenkin muutama heikkous kehiksessä nousee esille. Näitä ovat muun muassa suoritusteho, sovelluksen saaminen verkkoon, legacy-järjestelmän kanssa kommunikointi sekä sovelluskehiksen tuoma rajoittuneisuus [RRAD].

Sopiiko suunniteltu arkkitehtuuri järjestelmälle? Kehys ja kehiksestä johdettu MVC-mallinen arkkitehtuuri sopii todella hyvin tekemälleni järjestelmälle. Se on todella helposti muokattavissa sekä erilaisten näkymien/tietokannan vaihtaminen onnistuu hyvinkin nopeasti.

Suurimmat laadulliset vaatimukset järjestelmälleni olivat alussa, että se on nopea saada pystyyn, nopeaa kehittää ja monipuolinen jatkoon kannalta. Sovelluskehys tarjosi mielestäni nämä kaikki ja sivussa tuli muutama laatuviipukin.

3.1 Hyvät ja huonot puolet

Hyvät puolet kehiksessä ovat selkeästi kehittämisen nopeus sekä konventioiden tuoma järjestelmien virtaviivaistaminen, kun taas huonot puolet tulevat suurimmalta osin konventioiden pakollisuudesta sekä Ruby-kielestä.

3.1.1 Hyvät puolet

Kehiksen hyvät puolet tulevat melkein automaattisesti noudattamalla jopa toisinaan tiukkoja Railsin konventioita. Hyviin puoliin lukeutuvat muun muassa konventioiden kautta tuleva REST-rajapinta; nopea kehitys, joka sopii hyvin Agile-projekteihin; konventiot itsessään, jotka auttavat kehittäjiä siirtymään helposti Rails projektien välillä; kielenä Rubyn itsedokumentoiva tyyli sekä useimmat kehikseen liitettävät ulkopuoliset kirjastot, jotka ovat suurimmalta osin avointa lähdekoodia [RRWI].

Rails on tullut suosituksi myös startup-yrityksissä. Tähän on ollut syynä pääosin kehiksen: konventiot, kehys on käytännössä ilmainen kirjastoja myöten, erittäin hyvä PaaS-tuki (Platform as a Service), gem-ideologia (riippuvuudet yhdessä tiedostossa ja niiden helppo hallinta) sekä Rails on alustana kasvava ja reagoi uusiin tarpeisiin web-ympäristössä jatkuvasti [WSUR].

3.1.2 Huonot puolet

Suoritustehon heikkous Ruby on itsessään tulkattava kieli, joten suoritusteholtaan se häviää käännettävälle kielille esimerkkinä Java Spring. Toisaalta Ruby on Rails -kehiksen teho tulee tuottavuuden nopeudesta, mutta tuottavuus ei ratkaise yhtä Railsin suurinta ongelmaa; erittäin huonoa tukea säikeistämiseen, joka johtuu Rubyn globaalista tulkitsemisen lukitsemisesta (global interpreter lock) [MMRI]. Tämä tuottaa paljon ongelmia, jos järjestelmän yhtäaikaista käyttäjämäärä kasvaa suuressi, koska säikeistämisen avulla ei pystytä tasapainottamaan syntyneitä kuormaa (load balance). Täten Ruby on Rails soveltuu melko huonosti sovelluksille, jossa nopeus on oltava suuri myös silloin, kun käyttäjämäärät ovat suuria. Skaalautuvuus on myös tuottanut paljon puheita sekä puolesta, että vastaan [WDTS].

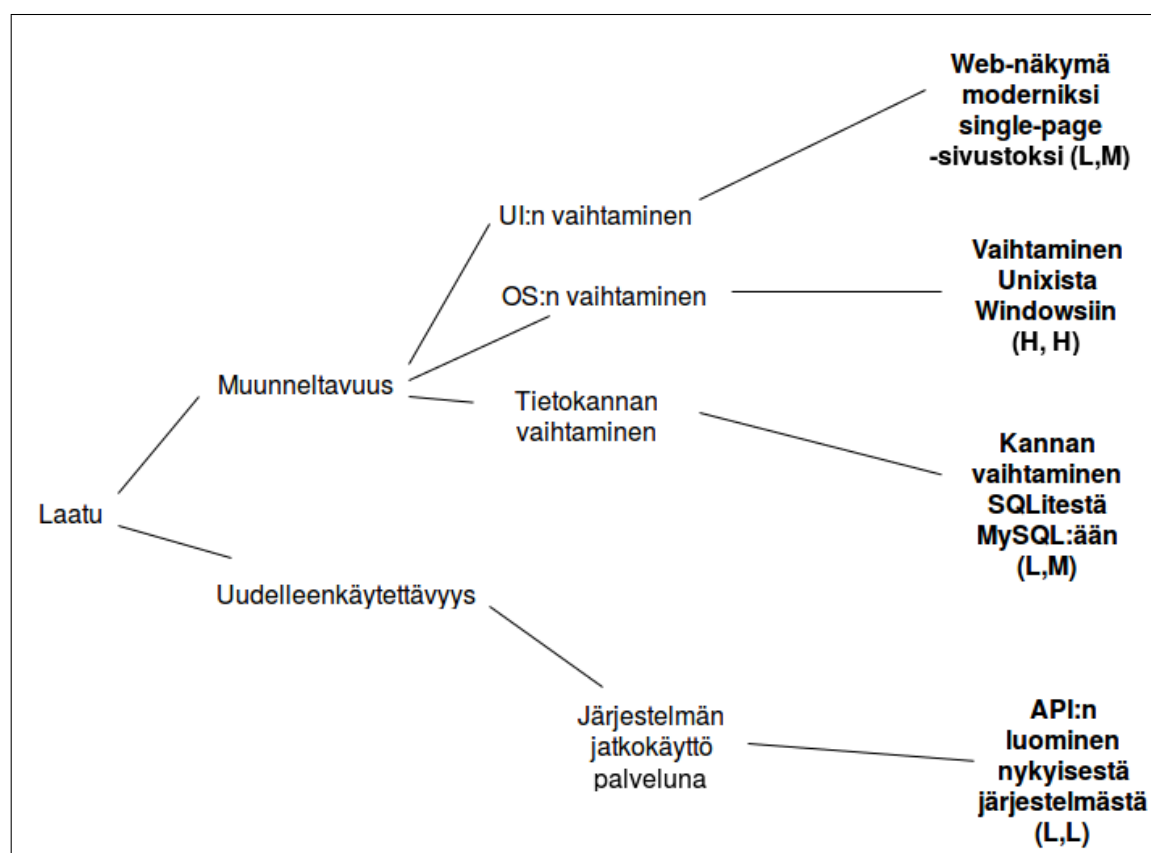
Sovelluksen vaikea tuotantoon saattaminen Rails-kehys vaatii paljon esiasennusta, koska esimerkiksi tavallisissa Linux-jakeluissa ei edes ole Ruby-kirjastoa valmiina. Tämän päälle pitää asentaa itse kehys ja se vie suhteellisen paljon enemmän tilaa ja aikaa, kuin muut paljon kevyemmät kehikset [RRWI]. Tästä johtuu suhteellisen huono palvelimien tuki Ruby on Rails-kehikselle ja aikaa saa kulumaan palvelimen säätämisessä Railsia varten.

Rajoittuneisuus Rails painottaa sisäisesti eleganttia syntaksia, tuotteliaisuutta sekä ylläpidettävyyttä nopeuden ja resurssien hallinnan kustannuksella. Nämä on toteutettu suurimallta osin konventioiden avulla, joten kehys saattaa tuntua kehittäjille todella rajoittavalta. Vaikka konventioista voi poiketa, niin jokainen poikkeama mahdollistaa potentiaalisen virheen myöhemmin järjestelmän elinkaareissa. Konventiot myös rajoittavat legacy-järjestelmien kanssa toimimista ja varsinkin legacy-tietokannan kanssa työskentely vaatii kehiksessä jopa todella paljon konfigurointia [URWL].

3.2 Arviointi

Kehystä arvioidessa keskitytään tässä raportissa kahteen laadulliseen tekijään: muunneltavuus sekä uudellenkäytettävyys. Nämä ovat myös tekijöitä, jotka tulevat esille melkein jatkuvasti Ruby on Rails-kehyksessä ja tuntuu, että ne ovat myös asioita joita kehys haluaa korostaa. Tarkastelussa kuitenkin huomataan tiettyjä riskejä, joita kehysten käyttö ja nykyiset arkkitehtuuriset ratkaisut tuovat mukanaan.

Laatupuu Muodostetut laatuskenaariot ovat luotu laatupuun avulla. Osa niistä on valittu mittaamaan järjestelmän riskialttiita kohtia, kun taas osa on todella yleisiä tapauksia, joita saattaa tulla eteen kehysten käytössä.



Kuva 9: Skenaariot sijaitsevat oikealla ja jokainen on arvioitu niin, että ensimmäisenä on suluissa skenaarion vaativuus ja sen jälkeen arvioitu työmäärä. L = Low, M = Medium, H = High.

3.2.1 Muodostettujen skenaarioiden analysointi

Web-näkymä moderniksi single-page -sivustoksi Näkymän vaihtaminen on kehyksessä todella helppoa kuten aiemmin on mainittu, mutta vain jos näkymän rakenne noudattelee perinteisempää web-sivustoa. Nykyaikaisen yhden sivun näkymän tekeminen vaatii paljon Javascript-yhteensopivuutta ja kehykseltä standardisointia JSON-rajapinnoille [IRRU]. Tätä kuitenkin hankaloittaa hieman Rails-kehys ja sen vahva MVC-patterni. Kuitenkin tähänkin ongelmaan löytyy valmiiksi ulkoinen kirjasto (gem), jonka avulla ongelma saadaan ratkaistua. Tästä syystä skenaarion vaativuus on matala, mutta sen toteutuminen vaatii kuitenkin ylimääräistä työtä.

Vaihtaminen Unixista Windowsiin Skenaario saattaa tulla eteen jo esimerkiksi silloin, kun projektissa on yksi kehittäjä, joka käyttää Windows-käyttöjärjestelmää. Jotta kehittäjä pystyisi jatkamaan projektia, niin tarvitaan todella suuri määrä konfigurointia projektiin Windowsia varten, koska suurin osa Rails-kirjastoista on tehty Unix-järjestelmää varten. Tämä johtaa siihen, että harva ulkoinen kirjasto toimii enää ja jos projekti käyttää paljon ulkoisia kirjastoja, niin melkein jokaiselle on löydettävä vaihtoehto Windowsia varten. Tämä tuottaa todella paljon ylimääräistä työtä ja sen takia tälle skenaariolle on annettu korkeimmat mahdolliset arviot.

Kannan vaihtaminen SQLitestä MySQL:ään Tietokannan vaihtaminen onnistuu Rails-kehyksellä jopa vain muutamalla komennolla [CRRÄ], mutta vain jos vanha kanta on seurannut kehysten konventioita. Kehys on varautunut toimimaan useilla eri relaatiotietokannoilla, mutta esimerkiksi dokumenttikannat vaativat jo ulkoisia kirjastoja. Kuitenkin SQLite on hyvin lähellä MySQL-tietokantaa, joten vaihtaminen onnistuu erittäin kivuttomasti. Vaihtoa helpottaa myös aiemmin mainittu ActiveDirectory-patterni, joten kannan vaihto ei vaadi yhtään ylimääräistä sisäistä konfigurointia esimerkiksi kyselyiden näkökulmasta. Tämän takia vaativuus on skenaariossa matala ja työmäärä keskitasoa. Työmäärä saattaa nousta juuri konventioista poikkeamalla.

API:n luominen nykyisestä järjestelmästä Uudelleenkäytettävyyteen liittyvä skenaario on helppo toteuttaa Rails-kehyksestä, koska Rails tukee erittäin hyvin JSON-tietomuotoa sisäisesti. Pelkästään noudattamalla konventioita saa luotua automaattisesti JSON-rajapinnan järjestelmästänsä. Sitä muokkaamalla ja muutamalla ulkoisella kirjastolla saa luotua erittäin helposti API:n järjestelmää varten [RFAP].

Suurin työ tulee pelkästään API:n suunnittelusta ja muusta kehyksen ulkopuolisesta työstä.

3.2.2 Skenaarioista johdetut päätelmät

Suurin riski kehyksessä on alustan vaihtaminen, koska kehys on sidottu hyvin vahvasti siihen käyttöjärjestelmään, jolla järjestelmän kehitys on aloitettu. Tämä heikentää todella paljon muokattavuutta, koska esimerkiksi jos järjestelmä täytyy siirtää eri palvelimien välillä, niin on otettava erittäin tarkasti huomioon millä käyttöjärjestelmällä palvelin pyörii; varsinkin jos kyseessä on Windows. Tästä johtuvaa muunneltavuuden huonontumista voidaan estää muunmuassa kartoittamalla sovellusalueita tarkasti, ottamalla etukäteen selvää ulkoisten kirjastojen tuista tai käyttämällä mahdollisimman vähän ulkoisia kirjastoja.

Vahvaa MVC-patternia kehyksessä voidaan pitää herkkyysskohtana, koska niin moni kehyksen tarjoamista laadullisista ominaisuuksista perustuu tähän. Siitä poikkeaminen johtaa ongelmiin kehyksen konventioiden kanssa ja saattaa huonontaa merkittävästi järjestelmän yleistä muunneltavuutta. Konventioista luopuminen taas heikentää merkittävästi tätä laatuominaisuutta, mutta toisaalta konventiot voidaan laskea kuuluvan myös tasapainottelukohtaan, koska ne edistävät muunneltavuutta, mutta muunneltavuuden parantaminen heikentää suorituskkyä. Tästä esimerkkinä ActiveRecord joka luo joustavan rajapinnan tietokantaan, mutta, jonka muodostamat käskyt saattavat joskus olla hyvinkin suuria ja hitaita [RRPT].

On kuitenkin huomattavaa, että kaikki muut skenaariot ovat arvioiltaan alhaisia, joten sovelluskehyksen käytöstä on erittäin paljon hyötyä, jos tietyt skenaariot ovat todennäköisiä järjestelmän elinkaareissa. Esimerkiksi tietokannan vaihtaminen saattaa tulla esille vasta myöhemmässä vaiheessa järjestelmän elinkaarta, jolloin sen vaihtaminen voi olla jollain muulla sovelluskehyksellä kallista ja hidasta. Ruby on Rails -kehyksellä se onnistuu erittäin nopeasti, mutta vain jos määriteltyjä konventioita on noudatettu jo aikaisemmankin tietokannan kanssa.

Lähteet

- ARP Active Record -pattern. https://en.wikipedia.org/wiki/Active_record_pattern. [28.12.2015]
- RRWI Ruby on Rails: What It Is and Why We Use It For Web Applications. <http://bitzesty.com/2014/01/10/ruby-on-rails-what-it-is-and-why-we-use-it-for-web-applications/>. [08.1.2016]
- MMRI Multithreading in the MRI Ruby Interpreter. <http://www.csination.com/2014/10/10/multithreading-in-the-mri-ruby-interpreter/>. [22.01.2015]
- RHTML Rails and HTML - RHTML. <http://www.tutorialspoint.com/ruby-on-rails/rails-and-rhtml.htm>. [04.1.2016]
- AR Active Record. <http://api.rubyonrails.org/classes/ActiveRecord/Base.html>. [02.1.2016]
- IRRU Is Ruby on Rails useful for a single page application? <https://www.quora.com/Is-Ruby-on-Rails-useful-for-a-single-page-application>. [11.1.2016]
- RFAP Rails for API only applications. <https://github.com/rails-api/rails-api>. [11.1.2016]
- ROR RoR - Ruby on Rails. <http://rubyonrails.org/>. [28.12.2015]
- RARN Ruby and Rails Naming Conventions. <http://itsignals.cascadia.com.au/?p=7>. [19.01.2015]
- RRAD Ruby on Rails Architectural Design. <http://adrianmejia.com/blog/2011/08/11/ruby-on-rails-architectural-design/>. [30.12.2015]
- RRPT Ruby on Rails Performance Tuning. <https://hackhands.com/ruby-rails-performance-tuning/>. [11.1.2016]
- URWL Using Rails with a legacy database schema. <https://schneide.wordpress.com/2014/03/10/using-rails-with-a-legacy-database-schema/>. [07.1.2016]

WDTS	Why Do They Say Rails Doesn't Scale? http://codefol.io/posts/why-do-they-say-rails-doesnt-scale . [07.1.2016]
RDWS	Ruby Default Web Server. https://devcenter.heroku.com/articles/ruby-default-web-server . [04.1.2016]
CRRA	Convert a Ruby on Rails app from sqlite to MySQL? http://stackoverflow.com/questions/1670154/convert-a-ruby-on-rails-app-from-sqlite-to-mysql . [10.1.2016]
WEBR	WEBrick. https://en.wikipedia.org/wiki/WEBrick . [04.1.2016]
WSUR	Why Startups Use Ruby on Rails? http://www.nascenia.com/why-startups-use-ruby-on-rails/ . [08.1.2016]
YAPI	Yandex - Translator API. https://tech.yandex.com/translate/ . [07.11.2015]