

Ruby on Rails -sovelluskehys; case: Translator

Kristian Wahlroos - 014417003

Helsinki 5.1.2016

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Sisältö

1	Johdanto	1
1.1	Translator -projekti	1
1.2	Ruby on Rails -sovelluskehys	3
1.3	Arkkitehtuuriset tyylit	5
2	Yleisarkkitehtuuri ja keskeisimmät variaatiopisteet	7
2.1	Kehyksen erikoistaminen sovelluskohtaisesti	11
2.2	Sekvenssikaavio	11
3	Kehyksen ja sovelluksen arviointi	13
3.1	Hyvät ja huonot puolet	13
3.2	ATAM-arviointi	14
	Lähteet	15

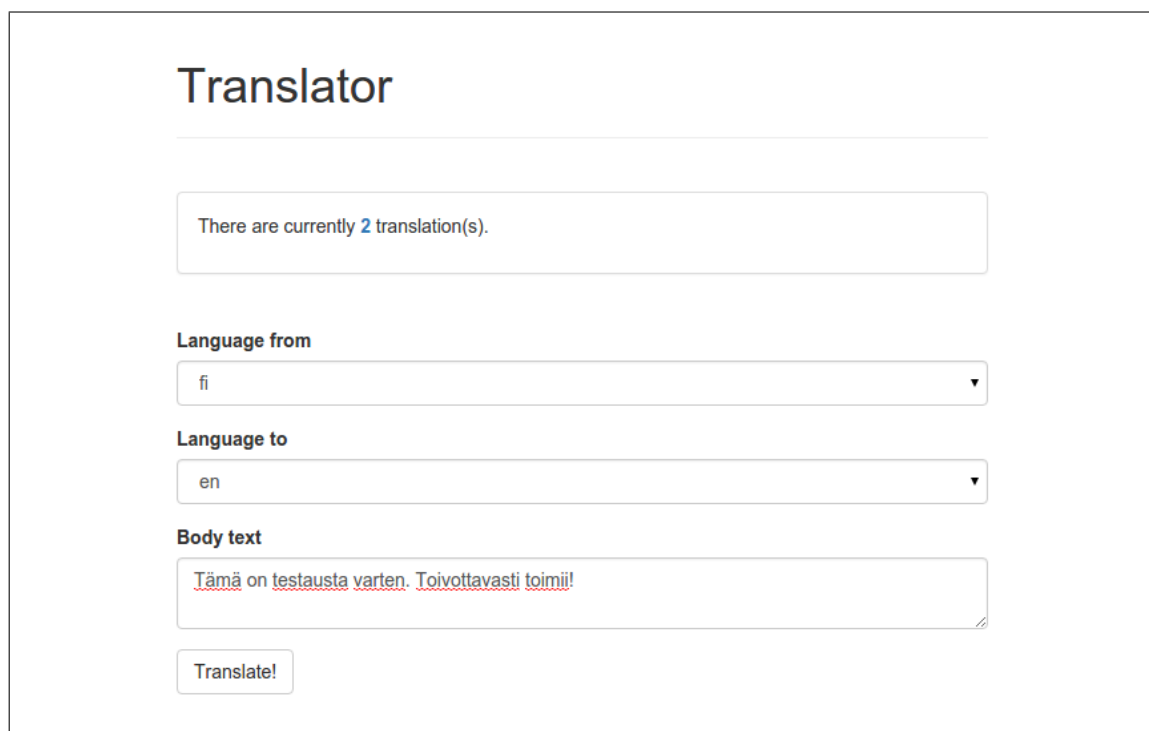
1 Johdanto

Tämän harjoitustyön tarkoituksena on tarkastella Ruby on Rails -web sovellushystä ja varsinkin sen arkkitehtuuria tekemäni esimerkkisovelluksen kautta. Teke-mäni esimerkkisovellus 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, vaan itse keskittyminen tapahtuu sovelluksen pinnan alle kuten arkkitehtuuriin ratkaisuihin joita Ruby on Rail-kehys tuo mukanaan.

1.1 Translator -projekti

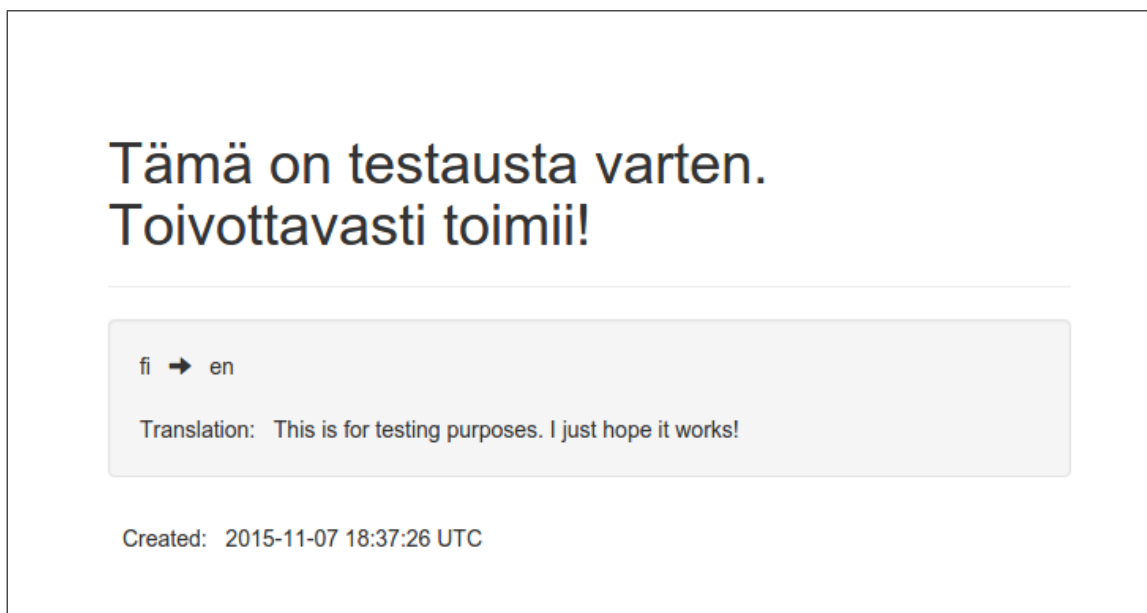
Sovellukseneeni viittaa tästä alkaen nimellä *Translator*.

Translatorin pääidea on siis nimensä mukaisesta kääntää annettu teksti kieleltä toi-selle. Käännöksen logiikka hoidetaan ulkopuolista API:a käyttäen [YANDEX], 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 message box that says "There are currently 2 translation(s)". Below this, there are two dropdown menus: "Language from" with "fi" selected and "Language to" with "en" selected. Below these, there is a text input field labeled "Body text" containing the text "Tämä on testausta varten. Toivottavasti toimii!". At the bottom, there is a button labeled "Translate!".

Kuva 1: Käyttäjä haluaa kääntää tämän 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:

Root (/) → tallennettu käännös (/translations/:id, jossa :id korvautuu tietokantaan tallennetun tietueen id:llä) → root (/) → 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 näkyvimpiä 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 tyylejä ei sovelluskehyksestä suoraan löydy, koska usealla komponentilla ei suoraan löydy vastaavuutta muista tyyleistä MVC:n lisäksi (MVC lasketaan tässä enemmänkin patterniksi), koska niiden vastuut ovat hieman laajempia. 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.

ActiveRecord itsessään toteuttaa Active Record-patternin [AR], joka määrittelee, että miten luokat ja tietokanta kuvautuu toisilleen. Rubyssä 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 Controllerin 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 ActionController:ksi Ruby on Railsissä ja se vastaa koko järjestelmään kohdistuvien pyyntöjen reitittämisestä sekä yleisestä datan välityksestä. Omassa järjestelmässäni TranslationController vastaa kaikesta käännöksiin liittyvästä toiminnallisuudesta, kuten esimerkiksi aloitussivun näyttämisen logiikasta. Railsissä 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 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 saanut automaattisesti käyttöön seuraavat osoitteet:

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 samalla sovelluskehityksellä tuotettujen kanssa. Esimerkiksi nimeämiseen liittyvät konventiot (projektini tietokantataulussani 'translations' sijaitsevat kaikki Translation-mallin ilmentymät, kontrolleri nimeltä 'translations_controller' huolehtii näiden mallien päivittämisestä sekä oikeiden näkymien näyttämisestä) ovat erittäin näkyvässä roolissa ja osittain jopa pakollisiakin.

Tämän avulla sovelluskehys osaa automaattisesti nimeämisten perusteella ohjata sovelluksen toimintaa haluttuun suuntaan, jolloin tarve konfiguraatio-tiedoistoille vähenee. Ruby on Rails siis pinnan alla automaattisesti päättelee nimien perusteella, että kenelle kyseiseen pyyntöön vastaamisen vastuu 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 tehtyt muutokset ja niiden peruutukset ('rollbackit').

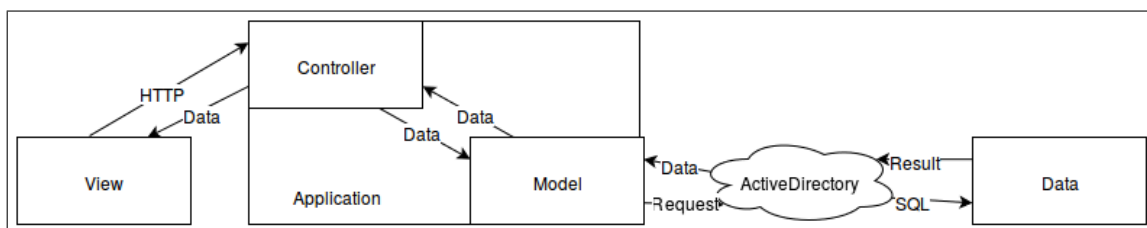
Toisaalta jo aiemmin mainittu ActiveRecord vähentää toistoa, koska kehittäjien ei tarvitse kirjoittaa erikseen ylös tietokantatauluja mallien määrittelytiedostoihin, vaan Ruby on Rails hoitaa kaiken tietokantalogiikan nimeämiskonventioiden avulla.

1.3 Arkkitehtuuriset tyylit

Kuten aiemmin mainittu, niin Ruby on Railsistä ei suoraan löydy juuri niinkään yhtä tiettyä tyyliä, jota kehys noudattaisi, mutta toisaalta 3-taso- malli on melko lähellä sitä.

Vastuut ovat siinä jaettu niin, että esimerkiksi tarkasteltessa kehystä tavallisen 3-taso -arkkitehtuurityylin silmin, erottuu joitain melko erilaisiakin vastuita. Näkömätasolla tosin vastuu on aika selkeä: näkymistä huolehtivat HTML-tiedostot, mutta sovellustasosta huolehtii mallit sekä kontrollerit, jotka saavat pyyntöjä UI:lta (HTML/HTTP). Datataso Railsissä koostuu ActiveDirectorystä, tietokannasta sekä varsinkin malleista. Mallit siis kuuluvat periaatteessa sovellustason sekä datatason välimaastoon, koska toisaalta kaikki sovelluslogiikka sijaitsee niissä, mutta toisaalta ne toimivat myös abstraktiona tietokannan tauluille ja koko tietokannan muokkaus tapahtuu suoraan mallien kautta.

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 omaan tietokantatauluun sekä hakevat tietoa kontrollerien käyttöön. Ne tarjoavat kontrollereille muokkaus/haku-rajapinnan tietokantaan. Tämä kyseinen kommunikaatio tietokantatauluun tapahtuu aina ActiveDirectoryn kautta, joka muuttaa pyynnöt kyseisen tietokannan osaamalle kielelle (usein SQL).

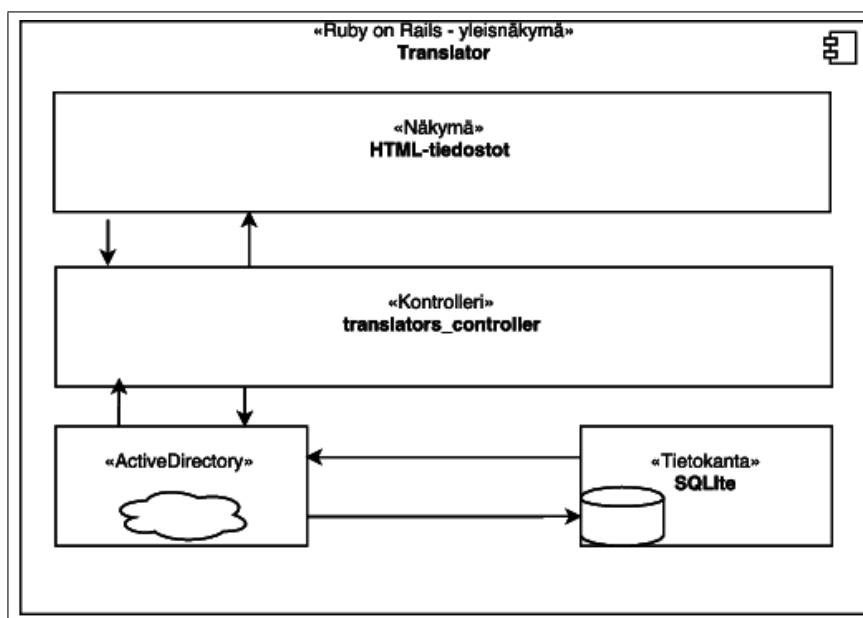


Kuva 4: Havainnekuva kommunikaatiosta.

2 Yleisarkkitehtuuri ja keskeisimmät variaatiopis- teet

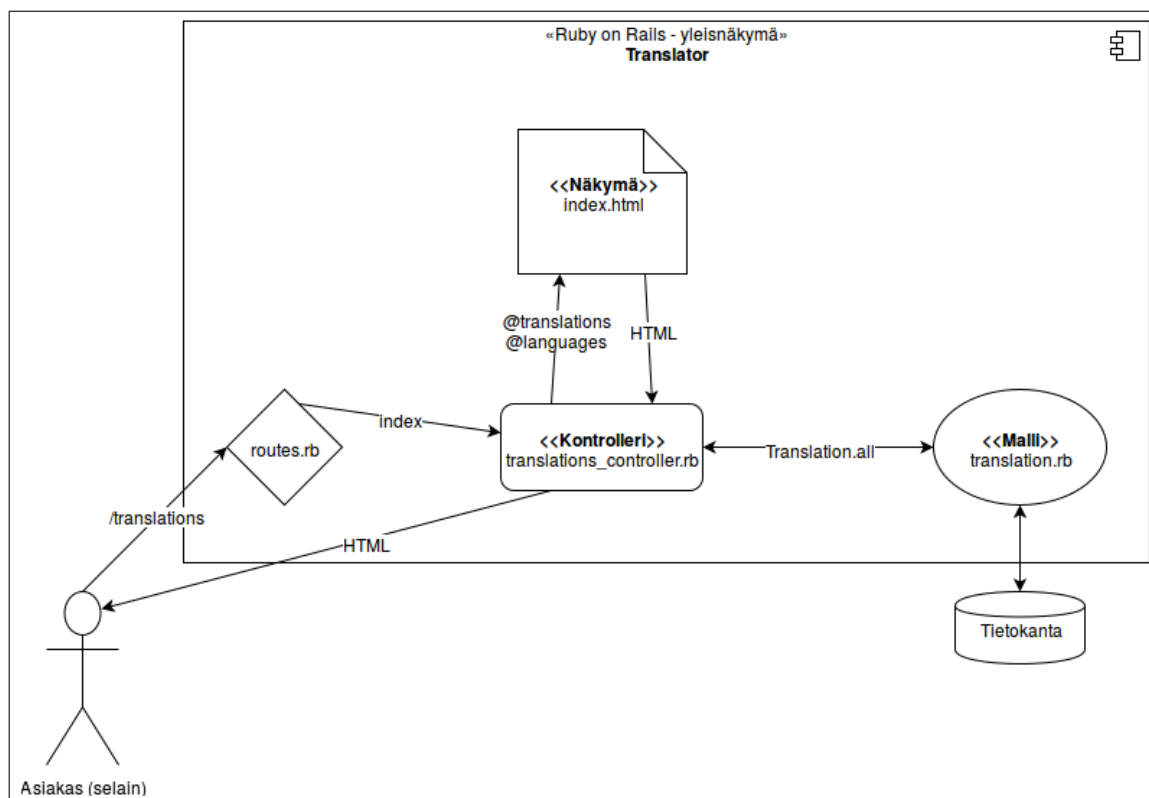
Ruby on Rails toteuttaa yleiseltä arkkitehtuuriltaan erittäin vahvasti jo aiemmin mainittua Model View Controller -tyyliä, mutta kehystä voidaan tarkastella myös perinteisemmän 3-taso -tyylin kautta.

Siinä vastuidenjakko on selvästi jaettuna kolmeen eri kerrokseen: tiedon tallennukseen, käsittelyyn sekä näyttämiseen. Tallennuksesta vastaa yhdessä tietokanta sekä Railsin ActiveDirectory-toteutus. Tieto itsessään käsitellään kontrollereiden tasolla, jossa tapahtuu myös mallien muokkaus ja lopullinen UI-muotoinen tiedon esitys tapahtuu HTML-tiedostojen avulla. Kerroksia ei pysty mitenkään ohittamaan, jolloin datan tallennus suoraan näkymästä on esimerkiksi käytännössä mahdotonta.



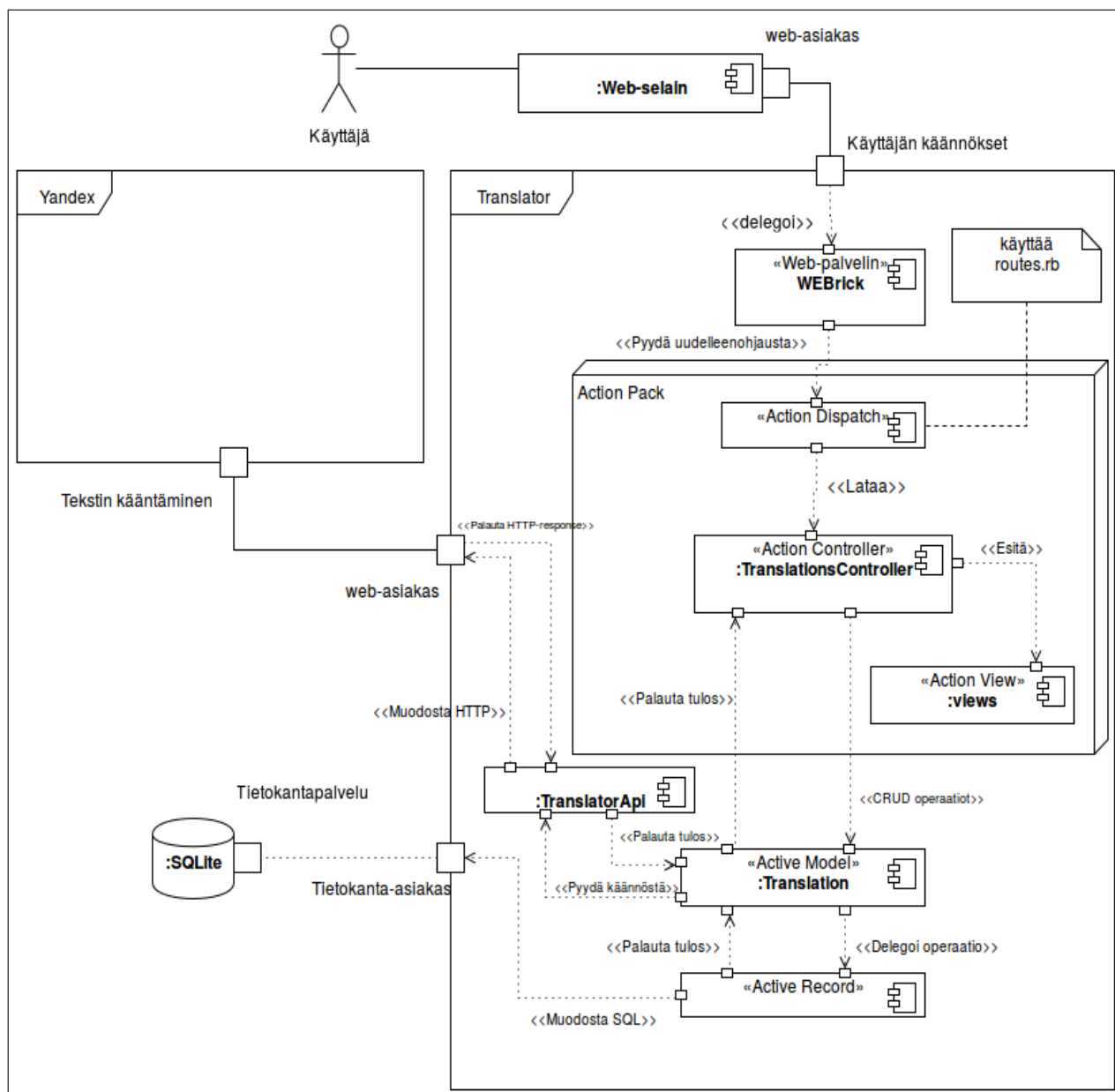
Kuva 5: Yksinkertaistettu arkkitehtuuri Translator-sovelluksesta 3-tasoisena tarkasteltaessa. Datan virtaus kuvattu nuolilla. Alin taso huolehtii tiedon tallennuksesta, keskimäinen tiedon käsittelystä ja ylin tiedon näyttämisestä.

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 näkymälle, joka luo halutun `HTML`-tiedoston. Lopuksi kontrolleri esittää (render) tämän dokumentin käyttäjän selaimelle.

Sisärakennemalli Seuraavassa Translator-järjestelmän komponentit.



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

Edellisestä kuvasta erottuu kaksi isoa järjestelmää, josta toinen on Translator-sovellukseni ja toinen Yandex-API, joka huolehtii kääntämisen logiikasta. Translator-sovelluksesta voidaan erotella seuraavat Ruby on Rails -kehityksen kannalta pakolliset komponentit.

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

Action Pack Tarjoaa kontrolleri- ja näkymäkerroksen 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.

Active Record Huolehtii datan muokkauksesta objektien kautta ja tarjoaa CRUD-rajapinnan tietokantaan (Create Read Update Delete). Jokainen operaatio tapahtuu aina luokan kautta, jolle löytyy vastaavuus tietokannassa. Luottaa paljon nimeämis-konventioihin muualla sovelluksessa, jolloin erinäisiä konfiguraatiotiedostoja ei tarvita.

2.1 Kehyksen erikoistaminen sovelluskohtaisesti

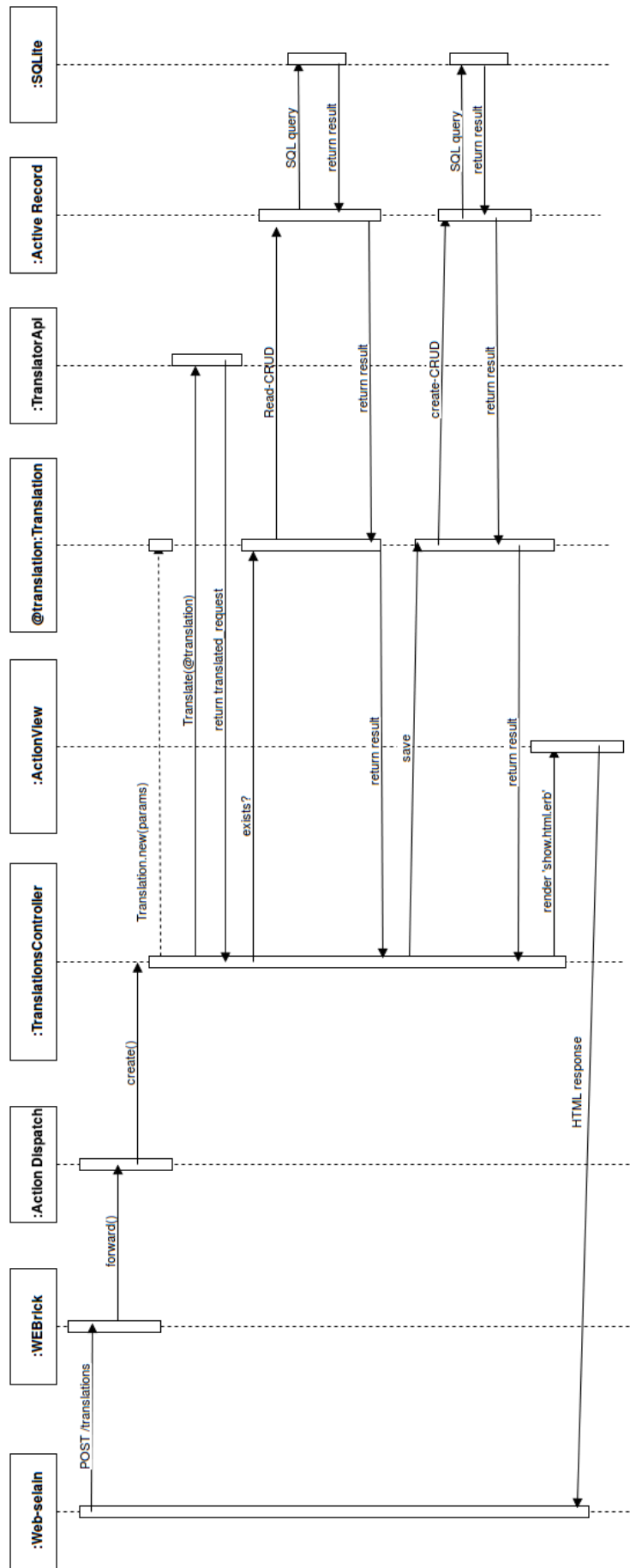
On huomattavaa, että esimerkiksi tietokanta ei sisälly järjestelmän sisälle, vaan ulkopuolelle ja on näin helposti vaihdettavissa. Tämä on toteutettu aiemmin mainitun ActiveRecordin avulla. Myös näkymien vaihdettavuus on erittäin helppoa, koska Ruby on Rails-konventioiden mukaan vain nimeämisellä (ja 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. Tätä ei pysty mitenkään muuttamaan, koska haluan sovelluksen nimenomaan olevan käännös-toimintoa tukeva sivusto, joka pystyy tallentamaan aiemmin tehtyjä käännöksiä. Tämän vuoksi talletuslogiikalla, näkymillä, eikä ohjainkomponenteille ole kovinkaan tärkeää roolia järjestelmässä ja tästä syystä ne voidaan laskea vaihtuviksi osiksi variaation kannalta Ruby on Rails-kehyksessä.

Kuvassa 7. olen merkanut 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 määritelty. Ainoastaan `TranslatorApi` on täysin itsenäinen oma staattinen luokkansa, jota `Translation` käyttää hyväkseen muodostaessaan kyselyitä ulkopuoliseen Yandex-APIin.

2.2 Sekvenssikaavio

Seuraavassa 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!` (Kuva 1). Sekvenssikaavio alkaa tästä tapahtumasta, mutta siinä on abstrahoitu joitan osia pois (funktiokutsuja) pääosin sen takia, että kuvaa olisi helpompi tulkita.



Kuva 8: Sekvenssikaavio käännöksen tekemisestä ja tallentamisesta.

3 Kehyksen ja sovelluksen arviointi

Mihin perustuu. Omat kokemukset. Arvioidaan kehystä ja sen avulla toteutetun sovelluksen arkkitehtuuria.

Sovelluskehyksenä Ruby on Rails on erittäin tarkka sen omista käytänneistä, kun sitä verrataan esimerkiksi toiseen erittäin suosittuun backend-sovelluskehykseen NodeJS:ään. NodeJS tarjoaa samalla tavalla kehyksen 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.

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

Suoritustehon heikkous Interpreted, no threads,

Sovelluksen vaikea deployaus Ruby on Rails vaatii paljon esiasennusta + itsessään isohko järjestelmä. Ruby ei ole tuettuina koneissa automaattisesti. Rails pitää itsessään asentaa

Legacy-ongelmat Konventiot.

Rajoittuneisuus Konventiot.

Sopiiko suunniteltu arkkitehtuuri järjestelmälle?

Mikä vaihtoehtoisista arkkitehtuureista sopii parhaiten järjestelmälle ja miksi?

Miten hyvä tulee olemaan järjestelmän jokin tietty laadullinen ominaisuus, olettaen järkevä toteutustapa?

3.1 Hyvät ja huonot puolet

Lähdekirjallisuudessa mainittuja

3.2 ATAM-arviointi

pienehkö joukko (laatu-) skenaarioita jonkin laatuominaisuuden testaamiseksi - sekä odotettavissa olevia että järjestelmän rajoja määritteleviä (katso luentokurssin materiaalit arkkitehtuurin arvioinnista ja ATAM:sta)

Esitetään konkreettisia esimerkkitilanteita, joissa tietyt laatuominaisuudet tulevat esiin

Laatuominaisuuden testaaminen - muunneltavuus ja uudelleenkäytettävyys (saatavuus/muunneltavuus/uudelleenkäyt.) Arkkitehtuurin arvioiminen Low/-Medium/High (vaativuus + työmäärä)

Jos tarkastellaan muutettavuutta, skenaario käsittelee jotakin muutostarvetta järjestelmän evoluutiossa

Jos tarkastellaan uudelleenkäytettävyyttä, skenaario käsittelee jonkin uuden soveluksen tekemistä uudelleen käyttämällä järjestelmää (= tuoterunko)

(.json-rajapinta)

Laatupuu

Identifioidaan riskit, turvalliset ratkaisut, herkkyyskohdat ja tasapainottelukohdat.

Lähteet

- AR Active Record -pattern. https://en.wikipedia.org/wiki/Active_record_pattern. [28.12.2015]
- BPS98 Bray, T., Paoli, J. ja Sperberg-McQueen, C., Extensible Markup Language (XML) 1.0. W3C Recommendation 10-February-1998. <http://www.w3.org/TR/1998/REC-xml-19980210>. [18.1.2000]
- EMN01 Erkiö, H., Mäkelä, M., Nykänen, M. ja Verkamo, I., Opinnäytetyön ulkoasun malli. Tieteellisen kirjoittamisen kurssiin liittyvä julkaisematon moniste, Tietojenkäsittelyopin laitos, Helsinki, 2001.
- RHTML Rails and HTML - RHTML. <http://www.tutorialspoint.com/ruby-on-rails/rails-and-rhtml.htm>. [04.1.2016]
- RORAR Active Record. <http://api.rubyonrails.org/classes/ActiveRecord/Base.html>. [02.1.2016]
- ROR RoR - Ruby on Rails. <http://rubyonrails.org/>. [28.12.2015]
- RORAD Ruby on Rails Architectural Design. <http://adrianmejia.com/blog/2011/08/11/ruby-on-rails-architectural-design/>. [30.12.2015]
- RUBYSERVER Ruby Default Web Server. <https://devcenter.heroku.com/articles/ruby-default-web-server>. [04.1.2016]
- WEBRICK WEBrick. <https://en.wikipedia.org/wiki/WEBrick>. [04.1.2016]
- YANDEX Yandex - Translator API. <https://tech.yandex.com/translate/>. [07.11.2015]