

Ruby on Rails -sovelluskehys; case: Translator

Kristian Wahlroos - 014417003

Helsinki 30.12.2015

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Sisältö

1	Johdanto	1
1.1	Translator	1
1.2	Ruby on Rails -sovelluskehys	3
1.3	Tyylit	5
2	Yleisarkkitehtuuri ja keskeisimmät variaatiopisteet	7
2.1	Yleiskuva kehysrakenteesta	8
2.2	Kehyksen erikoistaminen sovelluskohtaisesti	8
2.3	Suunnittelumallit	8
2.4	Esimerkkitapaukset	8
3	Kehyksen ja sovelluksen arviointi	9
3.1	Hyvät ja huonot puolet	9
3.2	Laatuskenaariot	9
3.3	ATAM	9
	Lähteet	10

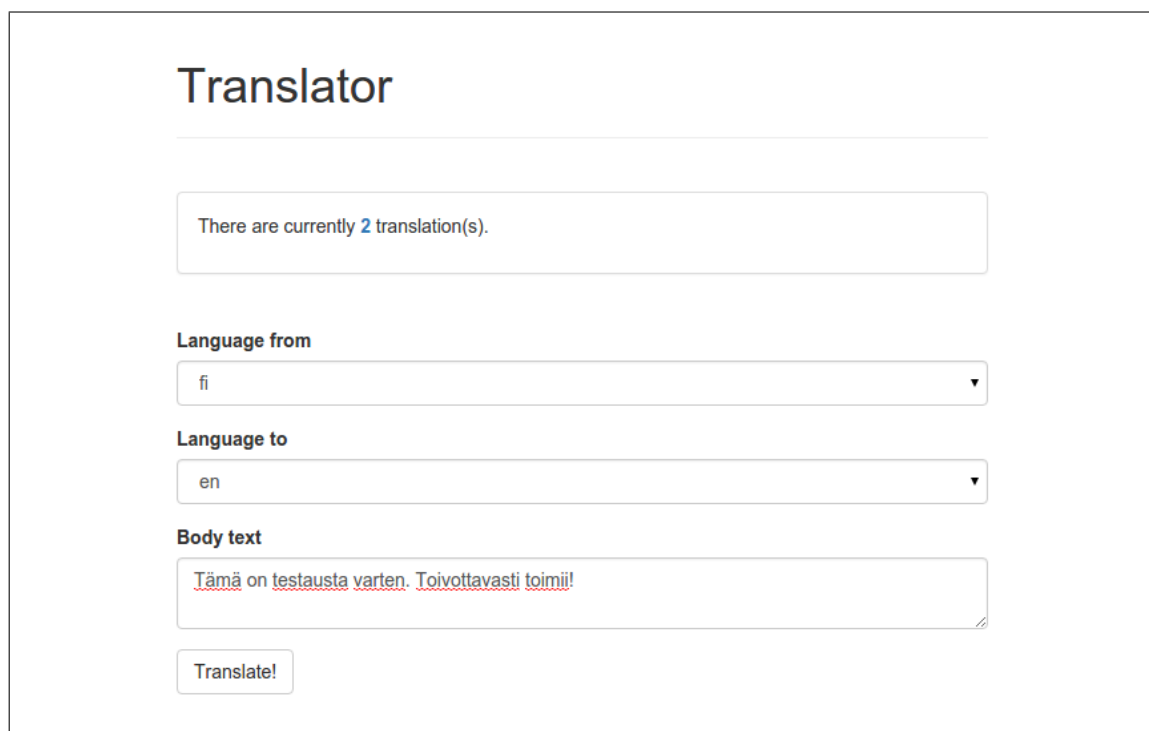
1 Johdanto

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

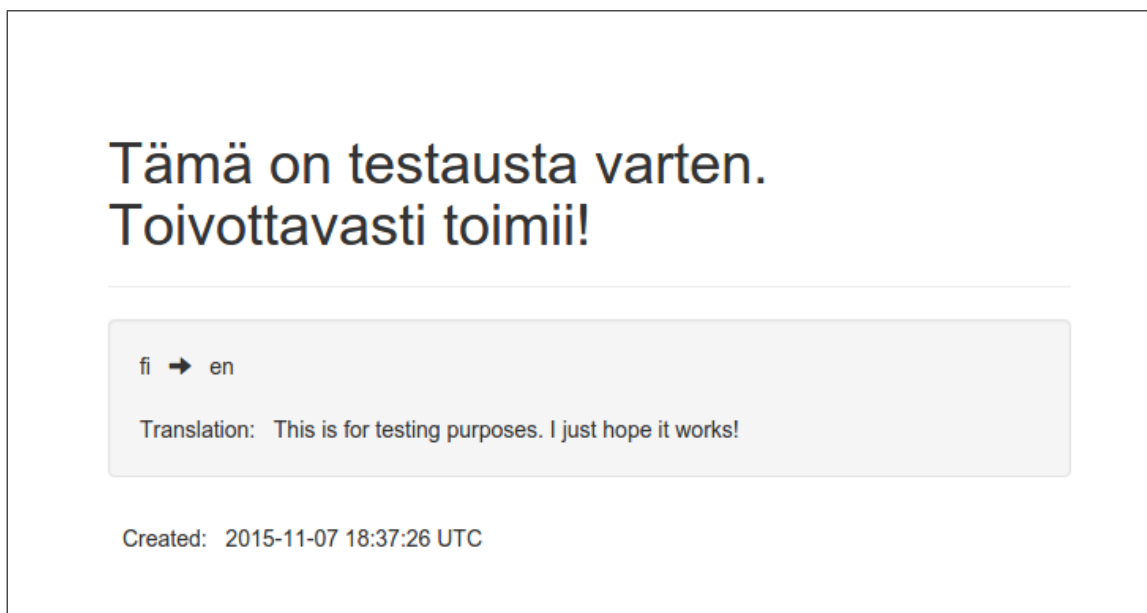
Sovellukseneeni viittaaan tästä alkaen nimellä *Translator*.

Translatorin pääidea on siis nimensä mukaisesta kääntää kieleltä toiselle. Käännök-sen logiikka hoidetaan ulkopuolista API:a käyttäen [YANDEX], joka hoitaa käytän-nössä kääntämisen sovelluslogiikan. Seuraavassa esitettävä kuvasarja esittää kääntä-misen 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 ActiveRecordissa säilytettävät 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 usein HTML-tiedosto sisältääkin upotettua Ruby-koodia. Tämä Ruby-koodi on suurimaksi 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 Railsissa 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. 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 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

Konventiot 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').

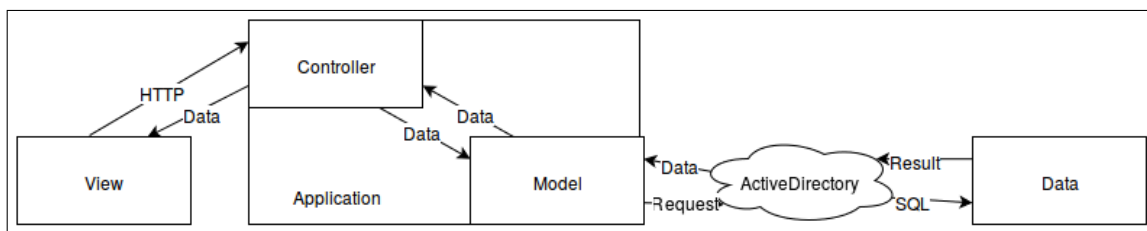
Toisaalta jo aiemmin mainittu ActiveRecord vähentää toistoa, koska kehittäjien ei tarvitse kirjoittaa erikseen ylös tietokantatauluja mallien määrittelytiedostoihin, vaan Ruby on Rails hoitaa kaiken tietokantalogiikan nimeämiskonventioiden avulla.

1.3 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.

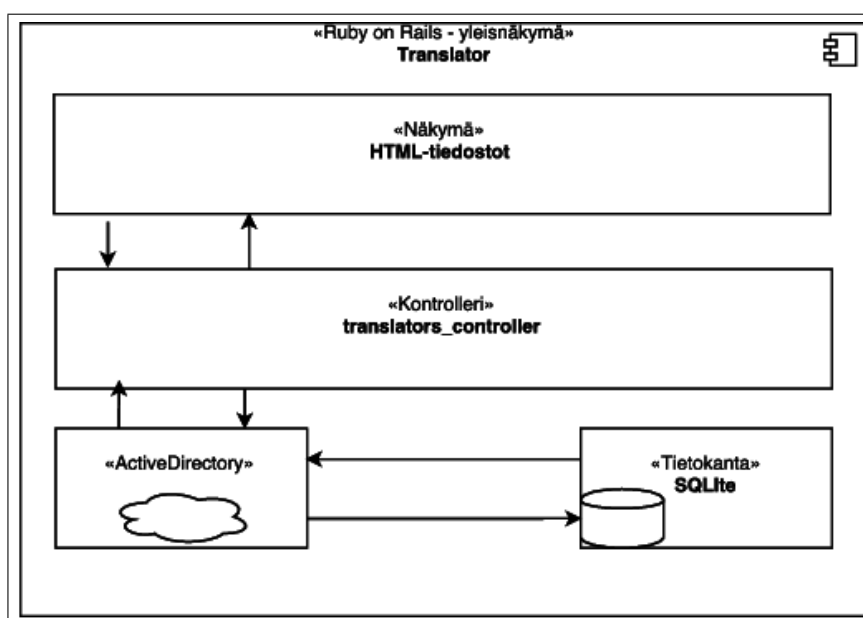
Asiakas-palvelin Kommunikointi kaikkien komponenttien välillä tapahtuu asiakas-palvelin-tyylin perusteella. Kontrollerit odottavat pyyntöjä näkymiltä ja tarjoavat tiedon näkymille mallien kautta. Mallit taas ovat suora linkki tietokantaan tehden muokkauksia sinne omaan tietokantatauluun sekä hakevat tietoa kontrollerien käyttöön.



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 N-tier -tyylin kautta. Siinä vastuidenjakoa on selvästi jaettuna kolmeen eri kerrokseen: tiedon tallennukseen, käsittelyyn sekä näyttämiseen. Tallennuksesta vastaa yhdessä tietokanta sekä Railsin ActiveDirectory-toteutus, tieto 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: Erittäin yleinen arkkitehtuuri Ruby on Rails -järjestelmissä. Datan siirtyminen kuvattu nuolilla.

MVC-mallisena tarkasteltaessa Ruby on Rails -sovelluskehys näyttäisi seuraavanlaiselta:

Luokkakaavio, sekvenssikaavio, stereotyytit.

- 2.1 Yleiskuva kehysrakenteesta
- 2.2 Kehyksen erikoistaminen sovelluskohtaisesti
- 2.3 Suunnittelumallit
- 2.4 Esimerkkitapaukset

3 Kehyksen ja sovelluksen arviointi

3.1 Hyvät ja huonot puolet

Lähdekirjallisuus

3.2 Laatuskenaariot

3.3 ATAM

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.
- ROR RoR - Ruby on Rails. <http://rubyonrails.org/>. [28.12.2015]
- YANDEX Yandex - Translator API. <https://tech.yandex.com/translate/>. [07.11.2015]