TARTU ÜLIKOOL
Arvutiteaduse instituut

Informaatika õppekava

Kaarel Tinn

Veebirakenduse loomine funktsionaalses programmeerimiskeeles Elixir

Bakalaureusetöö (9 EAP)

Juhendajad:
Kalmer Apinis

Tartu 2017

Veebirakenduse loomine funktsionaalses programmeerimiskeeles Elixir

Lühikokkuvõte

Käesoleva bakalaureusetöö eesmärk on tutvustada Elixiri programmeerimiskeelt ja sellel põhinevat Phoenix veebiraamistikku. Praktilise rakendusena luuakse programm ettevõttele E-Agronom OÜ. Rakendus suhtleb Eagronomi põhi- ja mobiilirakendustega, et salvestada kasutaja mobiilseadmelt saadud GPS andmeid.

Võtmesõnad: Elixir, Phoenix, GPS, eagronom, reaalaegne
CERCS: P170 - Arvutiteadus, arvanalüüs, süsteemid, kontroll

Building Web Application in Elixir

Abstract

The purpose of this thesis is to give brief introduction to Elixir programming language and Phoenix framework. From more practical side integration service for E-Agronom OÜ will be built. This service will be communicating with company's main program and mobile application in order to save GPS data sent from user's mobile device. Integration service will be built using Elixir and Phoenix.

Keywords: Elixir, Phoenix, GPS, eagronom, real-time
CERCS: P170 - Computer science, numerical analysis, systems, control

1. Sissejuhatus

Funktsionaalsed programmeerimiskeeled on kogumas aina rohkem populaarsust ning üha enam ettevõtteid võtavad kasutusele tehnoloogiaid, mis kasutavad funktsionaalse programmeerimise põhimõtteid. Elixir on dünaamiline funktsionaalne programmeerimiskeel, mis on viimasel ajal kogunud populaarsust.

E-Agronom on iduettevõte, mis teeb põllumajandusettevõtte haldamistarkvara. Üheks ettevõtte väljakutseks oli salvestada töötajate GPS-andmeid ja neid reaalajas ettevõtte omanikule kuvada. Seda soovitaksegi antud lõputöö raames implementeerida.

Elixir on sobiv lahendama paralleelust nõudvaid probleeme, kuna baseerub Erlangi virtuaalmasinal ning seetõttu pärib väga suure osa Erlangi võimekusest. Erlang on platvorm ehitamaks süsteeme, mis suudaksid samaaegselt teenindada mitmeid kliente olles samalajal äärmiselt veakindlad ja kergesti skaleeritavad [1]. Taolised mitte-funktsionaalsed nõudmised olid varem omased ainult vähestele programmidele nagu näiteks telekomi süsteemid. Erinevate tehnoloogiate arengutega on need jõudnud aga veebi, kus suur osa andmete käsitlust nõuab paralleelsust ja reaalaega. Elixir on sama võimekas kui Erlang ning kõike, mida saab teha Erlangis, saab teha ka Elixiriga. Elixir on palju tänapäevasema süntaksiga ning olles vabavaraline projekt leidub ka sellel aktiivne kasutajaskond, mis on ehitanud palju raamistike ja teeke, mis võimaldavad ehitada tänapäevaseid rakendusi [2].

Phoenix on raamistik, mis on mõeldud veebirakenduste ehitamiseks. Phoenixi peamine disaini printsiip on MVC, mida kasutavad enamus tänapäevaseid raamistike. Erinevalt teistest objekt-orienteeritud raamistikest on Phoenix funktsionaalne, mis võimaldab päringuga sissetulevaid andmeid väikeste funktsioonidega muuta. Kuna kõik andmed Elixiris on muutumatud, siis iga funktsiooni tulemuseks on täiesti uus struktuur, mitte ei muudeta rakenduse olekut implitsiitselt. See muudab paremini arusaadavaks miks, kus ja millal mingid muudatused päring-vastus tsüklis toimusid.

Käesoleva bakalaureusetöö eesmärk on tutvustada Elixiri programmeerimiskeelt ning Phoenixi veebiraamistiku. Teises peatükis antakse ülevaade Elixiri alustest ja peamistest ehitusblokkidest. Kolmandas peatükis implementeeritakse Phoenixi abil integratsiooni rakendus (vt Lisa 1) E-Agronomile, mis salvestab kasutaja mobiilsest seadmest tulevaid GPS-andmeid. Neljandas peatükis lisatakse esialgne reaalaja tugi kasutaja GPS-andmete jälgimiseks. Viies peatükk on kokkuvõte.

2. Elixir

Keele autor, Jose Valim, otustas luua Elixiri, kuna paljud populaarsed keeled ei ole väga head lahendama paralleelsust nõudvaid probleeme. Elixir on disanitud pidades silmas keele paindlikust. See tähendab, et lihtne on lisada keelele juurde konstruktsioone, mis pealiskaudselt vaadates tunduvad kui sisseehitatud. Tehnilises plaanis toetab Elixir makrode süsteemi ning polümorfismi läbi erinevate protokollide [3]. Samuti on olnud oluline, et keelega tuleksid kaasa võimalikult head tööriistad, mis kiirendaksid ning lihtsustaks programmeerija tööd.

Elixiri programmides on kesksel kohal andmed. Dave Thomas ütleb oma Elixiri tutvustavas raamatus "Programming Elixir 1.3" [4], et programmeerimine on andmete töötlemine ühest olekust teise ning siiamaani populaarne objekt-orenteeritud programmeerimine on vale lähenemine, sest programmeerides objektides, mõeldakse pidevalt objektide olekutele ja käitumisele, kuid sellega peidetakse implementatsiooni detailid klassidesse. See aga muudab andmete töötlemise hapraks, kuna ei ole teada, mida mingi objekt enda sisemuses teeb ja millises programmi osas olekut muudeb.

2.1 Andmetüübid

Oma olemuselt on Elixir dünaamilline. See tähendab, et andmetüübid pannakse paika käitusaegselt. Põhilised tüübid on keelde sisse ehitatud, kuid võimalik on ka neid oma äranägemise järgi juurde lisada. Elixir toetab täisarve, ujukomaarve, tõeväärtuseid, aatomeid, sõnesid, liste, ennikuid ning sõnastike. Kõik andmestruktuurid on muutumatud, mis tähendab, et kui tehakse mingeid operatsioone, siis tagastatakse täiesti uus andmestruktuur.

```
1                   # täisarv
1.0                 # ujukomaarv
true                # tõeväärtus
:foo                # aatom / sümbol
"string"            # sõne
["a", "b", "c"]     # list / järjend
{:a, :b, :c}        # ennik
%{a: 1, b: 2, c: 3} # sõnastik
```

2.2 Funktsioonid

Funktsioonid on Elixiris kõrgemat järku, mis tähendab, et funktsioone saab viidata muutujate kaudu ja anda argumendina teistele funktsioonidele. Funktsioonid tuleb defineerida moodulitesse, mis on funktsioonide kogumid.

```
defmodule Greeter do
  @default_name “stranger”

  def greet(name \\ @default_name) do
    IO.puts “Hello, #{name}”
  end
end

Greeter.greet                  # Hello, stranger!
Greeter.greet(“John”)          # Hello, John!
```

Üleval toodud koodilõik defineerib `Greeter` mooduli ning funktsiooni `greet`, mis kasutades standard teegis leiduvat `IO.puts/1` meetodit, väljastab ekraanile tervitava teksti. Kaldkriipsuga funktsiooni taga tähistatakse funktsiooni aarsust. Aarsus näitab mitu argumenti funktsioon nõuab. Antud näites teisel real defineeritud `@default_name` on mooduli atribuut, mis siinkohal täidab konstandi rolli. Sõned Elixiris toetavad interpolatsiooni, mis aitavad dünaamiliselt sõnesid koostada. Kommentaarid algavad `#` sümboliga. Funktsiooni tagastusväärtuseks on viimase avaldise väärtus.

Funktsionaalsele keelele kohaselt saab defineerida ka anonüümseid funktsioone, mida saab mugavalt teistele funktsioonidele argumendina anda. Anonüümsed funktsioonid tuleb defineerida võtmesõnade `fn..end` vahele.

```
sum = fn
  a, b -> a + b
end

sum.(1, 2)      # 3
```

Punkt muutuja nime taga funktsiooni väljakutsel aitab eristada anonüümseid funktsioone mitteanonüümsetest.

2.3 Sobitamine

Elixiri installeerimisega tuleb kaasa ka interaktiivne käsurea tööriist `iex`, mis võimaldab käsurealt hõlpsalt erinevaid keele konstrukutsioone järele proovida. Interaktiivse käsurea saab avada kirjutades käsureale `iex`. Samuti saab iga Elixiri rakenduse avada interaktiivse käsurea sessioonis, et testida rakenduse tööd ja saada kohest tagasisidet programmi töö kohta. Selleks tuleb rakenduse kaustas olles kirjutada terminali aknasse `iex -S mix`.

Elixir seab ohtu siiamaani teada-tuntud programmeerimise tavad [4:13]. Üheks olulisemaks muudatuseks on omistamise kaotamine ja asendamine mustrite sobitamisega. Elixiris ei omistata muutujatele väärtusi, vaid muutujad seotakse väärtustega läbi sobitamise. Tänu sobitamisele on tihti peale võimalik keerulised `if` ja `else` harud asendada deklaratiivsete sobitamise avaldistega.

```
iex> foo = 1
1
iex> 1 = foo
1
iex> foo = 2
2
iex> 1 = foo
** (MatchError) no match of right hand side value: 2
```

Sobitamise operatsioon tagastab alati väärtuse, milleks on paremal pool asuva avaldise väärtus. Kui vasakul pool asub muutuja, siis seotakse see paremal pool asuva väärtusega. Võrdusmärgi parem pool proovib end alati sobitada vasaku poolega. See tähendab, et paremal pool olevad avaldised väärtustatakse enne kui neid hakatakse vasaku poolega sobitama. Vasakut poolt ei väärtustata ning seega saab muutujat ka ümber siduda.

Sobitamine toimib ka keerulisemate andmestruktuuride pealt ning tihtipeale kasutatakse seda, et kindla struktuuriga andmeid “lahti pakkida”. Näiteks saab kolme-elemendilise järjendi kõik elemendid siduda muutujatega ühe avaldise abil.

```
iex> [a, b, c] = [1, 2, 3]
[1, 2, 3]       # a = 1, b = 2, c = 3
iex> [a, b, _] = [1, [2, 3], 4] # a = 1, b = [2, 3]
```

Sobitamise abil saame otsustada, millist funktsiooni implementsiooni rakendada. Näiteks võib anonüümsetel funktsioonidel olla rohkem kui üks funktsiooni keha.

```
iex> arvuta = fn
…> :+ , a, b -> a + b
…> :- , a, b -> a - b
…> tehe, _ , _ -> "Ei osanud #{tehe}'ga midagi peale hakata"
…> end
#Function<18.52032458/3 in :erl_eval.expr/5>
iex> arvuta.( :+ , 1 , 2 )
3
iex> arvuta.( :- , 1 , 2 )
-1
iex> arvuta.( :^ , 1 , 2 )
"Ei osanud ^'ga midagi peale hakata"
```

2.4 Andmestruktuurid

Listid võivad sisaldada erinevate tüüpidega elemente ning elemendid võivad korduda. Samuti on oluline elementide järjestatus. Oma implementatsiooni eripära tõttu on n-nda elemendi leidmiseks vaja läbida kõik eelnevad elemendid. Seepärast kasutatakse järjendi töötlemisel peamiselt rekursiooni.

```
iex> [head, tail] = [1, 2, 3]       # head = 1, tail = [2, 3]
iex> []                             # []
iex> [1]                            # [1]
iex> [1 | [2]]                      # [1, 2]
iex> [1 | [2 | [3 | []]]]           # [1, 2, 3]
```

Ennikuid kasutatakse tavaliselt erinevate andmetüüpide grupeerimisel. Tavaline on funktsioonist tagastada ennik, mille esimene element on aatom, ütlemaks, kas funktsiooni töö oli edukas või mitte.

```
iex> ennik = { :ok , 1 }
{ :ok , 1 }
iex> { :ok , value } = ennik
{ :ok , 1 }
iex> value
1
```

Sõnastike võtmed võivad olla eri tüüpi, kuid kui võtmed on aatomid, siis pakub Elixir lühemat süntaksit sõnastikuga opereerimiseks.

```
iex> map = %{ :nimi => "John Doe" , "vanus" => 26 }
%{:nimi => "John Doe" , "vanus" => 26 }
iex> map[ "vanus" ]
26
iex> map = %{ nimi: "John Doe" , vanus: 26 }
%{nimi: "John Doe" , vanus: 26 }
iex > map.nimi
"John Doe"
iex > map.vanus
26
iex > map[:amet]
nil
iex > map.amet
** ( KeyError ) key :amet not found in : %{nimi: "John Doe",
vanus: 26}
iex> %{ map | vanus: map.vanus + 1 }
%{nimi: "John Doe" , vanus: 27}
```

Elixiris leidub mugav viis andmete modelleerimiseks domeeni põhiselt. Selleks on sõnastikega sarnased struktid. Erinevad nad selle poolest, et neil on nimi ning kindlad vaikimisi väärtused eeldefineeritud võtmetele. Strukte saab defineerida ainult moodulis ning seda tehakse *defstruct/1* makroga. Strukt omandab mooduli nime.

```
defmodule User do
  defstruct first_name: “John”, last_name: “Doe”
end
```

```
> user = %User{}
%User{first_name: “John”, last_name: “Doe”}
> user = Map.put(user, :first_name, “Jane”)
%User{first_name: “Jane”, last_name: “Doe”}
```

Phoenixi veebiraamistik kasutab peamiselt strukte andmete modelleerimiseks.

2.5 Toruoperaator

Elixiris programmeerides on tähtsal kohal andmed ja nende töötlemine ühest olekust teise. Siin kohal leidub Elixiris mugav operaator, mis võimaldab järjestikuseid funktsioonide väljakutseid selgemalt väljendada.

```
iex> %{a: 1, b: 2, c: 3} |>
…> Enum.map(fn {k, v} -> v + v end) |>
…> Enum.each(&IO.puts/1)
2
4
6
:ok
```

`%{a: 1, b: 2, c: 3}` on sõnastik, mida asutakse töötlema. `Enum` moodul sisaldab endas funktsioone, mis võimaldavad sooritada erinevaid operatsioone erisuguste andmestruktuuride peal. Toruoperaator `|>` annab vasakul pool oleva väärtuse esimese argumendina paremal poolel asuvale funktsioonile. Niimoodi on võimalik tähendusrikkalt andmete transformeerimist väljendada.

2.6 Kontroll konstruktsioonid

Kuigi tänu sobitamisele ei kohta Elixiri programmides väga palju `if..else` lauseid on nad siiski keeles olemas. Kuna tegu on tegelikult Elixiri makroga, mitte keele päris oma enda konstruktsiooniga, võib mõelda `if` lausest, kui kahese aarsusega funktsioonist, mis saab argumentideks tõeväärtuse ja võtmesõnade listi kujul `[do: …, else: …]` ehk järgmised avaldised on samaväärsed.

```
iex> if true, do: 1, else: 2
1
iex> if(true, do: 1, else: 2)
1
iex> if true do
….> 1
….> else
….> 2
….> end
1
```

Elixiris on väärad väärtused ainult *nil* ja *false*. Kõik teised väärtused on tõesed.

Enam kohtab Elixiri programmides *case* ja *cond* lauseid. Esimene neist on sarnane paljudes teistes keeltes oleva *switch* lausega. *case* võimaldab meil võrralda väärtuse mustrit ja siis sobitamise abil otsustada, millist implementatsiooni käiku lasta.

```
case {:a, 2, “foo”} do
  {2, :a, “foo”} -> “No match!”
  {:a, 2, “foo”} -> “Match!”
  _ -> “Also won’t match.”
end
```

Esimene sobituv muster otsustab, milline implementatsioon välja kutsutakse ning mustreid vaadatakse läbi defineerimise järjekorras. Alati on mõttekas lisada viimase mustrina _, mis sobitub vaatamata vaatluse all oleva väärtuse mustrile. Kui ükski muster ei sobitu, siis viskub *CaseClauseError.*

Teine palju kasutust leidev tingimuslause on *cond*, mis võimaldab erinevate tõeväärtuste puhul valida. Sisuliselt on see kompaktne viis asendamaks pikki *if…else* lauseid.

```
cond do
  1 == 2      -> “Won’t run!”
  “1” == “2” -> “Also won’t run!”
  2 == 2      -> “Will run!”
end
```

3. Integratsiooni rakendus

3.1 E-agronom

E-agronom on veebipõhine rakendus haldamaks põllumajandusettevõtte tööd. E-agronom võimaldab põllumehel importida põldude andmed PRIA-st E-agronomi keskkonda ja hallata põlluraamatut, luua erinevaid viljavaheldusi, planeerida tulevase hooaja töid. Need on ainult vähesed võimalused, mida saab E-agronomi abil ettevõtte paremaks haldamiseks teha.

E-agronomil on olemas mobiilirakendus (vt Lisa 1), mis on mõeldud ettevõtte töötaja jaoks, kes täidab põldudel erinevaid töid. Mobiilirakenduses on võimalik näha töötajale määratuid töid, mis on varem ettevõtte omaniku poolt planeeritud. Töötaja näeb rakendusest töö tüüpi ning materjale, mis töö sooritamiseks vaja läheb. Lisaks näeb ta kaardi pealt enda ja põldude asukohti. Töötaja saab märkida töö tehtuks ja sisestada, kui palju tegelikult materjale kulus. Kogu see informatsioon kajastub E-agronomi süsteemis ja on koheselt ettevõtte omanikule nähtav. Taustal loeb mobiilirakendus seadme GPS-andmeid ning kui andmed muutuvad, saadab need koos autentimisvõtmega integratsiooni rakendusse, mis valideerib kasutaja andmed ja salvestab need eraldiseisvasse andmebaasi.

3.2 Implementatsioon

Integratsiooni rakendus on ehitatud kasutades Phoenixi veebiraamistiku. Ametlik Phoenixi koduleheküIg tutvustab seda kui raamistikku, millega on lihtne ja mugav ehitada tänapäevaseid veebirakendusi, kasutades funktsionaalseid, turvalisi ja tõsiselt võetavaid tehnoloogiaid [6]. Phoenix võimaldab meil mõelda veebirakendusest kui ühest suurest funktsioonist, mis saab argumendiks URLi ning tagastab selle põhjal vastuse. Tehniliselt saab Phoenix päringust andmed ja muudab need `%Plug.Conn{}` struktuuriks, mida siis erinevad väiksemad funktsioonid asuvad järge mööda transformeerima.

Uue Phoenixi rakenduse saab luua käsuga `mix phoenix.new <rakenduse nimi>`. Selle tulemusena luuakse esmane töötav rakendus.

Failid, mis on vajalikud veebirakenduse tööks asuvad `web` kaustas. Seal asuvad kontrollerid, mudelid, vaated ja mallid. Need on MVC põhilised ehitusblokid. Kontrollerid juhivad ja suunavad sissetulevate päringute andmeid. Mudelid defineerivad domeenispetsiifilised andmetestruktuurid ja sisaldavad vajalikku äriloogikat. Vaated sisaldavad funktsioone, mis tagastavad vajalikus formaadis andmed mallide jaoks. Mallid on tavaliselt lihtsad HTML või JSON failid, kuvamaks andmeid brauseris. Samuti leiab `web` kaustast `router.ex` faili, mis defineerib erinevad marsruudid (routes) ja konveierid (pipelines), mis on esimeseks kokkupuuteks päringutest tulevate andmete jaoks. Marsruut

on sihtpunkt, mille pihta saadetakse päringud. Konveierid sisaldavad erinevaid funktsioone, mis üksteise järel andmeid transformeerivad.

```
defmodule Integration.Router do
  use Integration.Web, :router

  pipeline :api do
    plug :accepts, ["json"]
    plug Integration.Auth
  end

  scope "/api", Integration do
    pipe_through :api

    post "/geodata", GeodataController, :geodata
  end
end
```

Eelpool sai mainitud, et Elixir toetab makrosid, mis on koodiosad, mida saab käivitada kompileerimisaegselt. See tähendab, et kompileerimisel peab olema meil ligipääs kasutatavatele makrodele. Direktiiviga *use* saame vajaminevad makrod mooduli kontekstis kättesaadavaks teha [7]. Märksõna *pipeline* abil on võimalik defineerida konveierit, mis teostab erinevaid transformatsioone enne kui päring jõuab marsruudini. Märksõna *scope* võimaldab sarnased marsruudid grupeerida ühise nimeruumi alla. Integratsiooni rakenduse marsruuter defineerib *POST /api/geodata* marsruudi, millele saab saata JSON formaadis andmeid, mida seejärel *GeodataController#geodata* funktsioon saab töödelda. Enne veel kui andmed jõuvad kontrollerisse, läbivad need *api* konveieri, mis filtreerib ainult JSON formaadis andmed ning valideerib kasutaja. Konveieri funktsioone kutsutakse pistikuteks (plugs), mis on Elixiri moodul või funktsioon, mille abil saab *%Plug.Conn{}* andmestruktuuri muuta.

Pistik *Integration.Auth* vastutab selle eest, et iga POST päring oleks tehtud valideeritud kasutaja poolt.

```
defmodule Integration.Auth do
  import Plug.Conn

  @core_api Application.get_env(:integration, :core_api)

  def init(opts) do
    opts
  end

  def call(%Plug.Conn{params: %{"user_credentials" =>
user_credentials}} = conn, _opts) do
    case validate_user_credentials(user_credentials) do
      {:ok, user_id} ->
```

```
        assign(conn, :user_id, user_id)
      {:error, reason} ->
        conn |> put_resp_content_type("application/json") |>
send_resp(401, reason) |> halt
    end
  end

  def validate_user_credentials(user_credentials) do
    case @core_api.validate_user(user_credentials) do
      {:ok, %{"success" => true, "user_id" => user_id}} ->
        {:ok, user_id}

      {:ok, %{"success" => false, "message" => message}} ->
        {:error, message}

      {:error, reason} ->
        {:error, reason}
    end
  end
end
```

Kõigepealt tuleb importida pistiku funktsionaalsus *import Plug.Conn* direktiiviga. Moodul pistik peab defineerima vähemalt kahte funktsiooni - *init/1* ja *call/2*. Funktsioon *init/1* initsaliseerib pistiku vajaminevate argumentidega. Funktsioon *call/2* viib aga läbi *%Plug.Conn{}* transformatsioone. Integratsiooni rakenduse puhul *init/1* funktsioon edastab pistikule kõik argumendid, mis kaasa antakse. Funktsioon *call/2* vaatab sobitamise abil, et *%Plug.Conn{}* sisaldaks *user_credentials* parameetrit. Kui *user_credentials* parameeter on olemas, siis proovib *validate_user_credentials/1* meetod seda valideerida, küsides E-agronomi põhirakenduse (vt Lisa 1) käest, kas *user_credentials* parameeter on valiidne. Valiidne tähendab siinkohal, kas leidub kasutaja, kes valideeritakse põhirakenduses *user_credentials* parameetris sisalduva võtmega. Kui jah, siis tagastatakse *{:ok, user_id}* ennik, kus *user_id* on kasutaja *id* põhirakenduses. Kui *user_credentials* ei ole valiidne, siis tagastatakse *{:error, message}* ennik ning kui päring põhirakendusse peaks ebaõnnestuma, siis annab sellest teada *{:error, reason}* ennik.

Suhtlust põhirakendusega haldab eraldi moodul *Integration.CoreAPI*, mis sisaldab üht ainsat funktsiooni *validate_user/1.*

```
defmodule Integration.CoreAPI do
  @url Application.get_env(:integration, :core_url)

  def validate_user(user_credentials) do
    body = %{"user_credentials" => user_credentials} |>
Poison.encode!
    headers = %{"Content-Type" => "application/json"}
    case HTTPoison.post(@url <> "/api/validate_user", body,
headers) do
      {:ok, %HTTPoison.Response{body: body}} ->
        {:ok, body |> Poison.decode!}
      {:error, %HTTPoison.Error{reason: reason}} ->
        {:error, reason}
    end
  end
end
```

Funktsiooni *validate_user/1* ülesanne on ehitada päringu jaoks vajalik andmestruktuur ning teha päring põhirakendusse. Põhirakenduse URL tuleb Phoenixi seadetest, kuna testimisel ei ole mõistlik teha päris HTTP päringuid, siis on vaja need välja *mockida.* Elixir teek *HTTPoison* võimaldab teha HTTP päringuid ja selle abil tehakse päring põhirakenduse */api/validate_user* marsruudi pihta. Sobitamise teel tehakse kindlaks, kas päring oli edukas või mitte ning tagastatakse vastav ennik.

Kui *user_credentials* sisaldab valiidset võtit, siis salvestatakse *call/2* funktsioonis kasutaja identifitsiseerimistunnus *%Plug.Conn{}* struktuuri. Vastasel juhul aga kogu edasine töötlus peatatakse ning tagastatakse HTTP 401 staatuskoodiga vastus päringu tegijale.

Järgmisena jõuab tööjärg juba kontrolleris oleva funktsioonini, mille peamine ülesanne on otsustada, kas andmed salvestada ning vastavalt selle, kas salvestamine õnnestus või ei õnnestunud, saadetakse saatjale vastus.

```
defmodule Integration.GeodataController do
  use Integration.Web, :controller

  alias Integration.Geolocation

  def geodata(conn, %{"lastPosition" => last_position}) do
    changeset = Geolocation.changeset(
      %Geolocation{},
      %{"geodata" => Poison.decode!(last_position), "user_id" =>
conn.assigns[:user_id]}
    )
    case Repo.insert(changeset) do
      {:ok, _geolocation} ->
        conn
        |> send_resp(200, "Geolocation saved.")
```

```
      {:error, _changeset} ->
        conn
        |> send_resp(300, "Geolocation not saved.")
    end
  end
end
```

Nagu ka *Integration.Auth* pistiku puhul, tuleb ka siin importida kontrolleritele vajalik funktsionaalsus direktiiviga *use*. Meetod *geodata/2* saab argumentideks *%Plug.Conn{}* struktuuri ja päringu parameetritest koosneva sõnastiku.

```
defmodule Integration.Geolocation do
  use Integration.Web, :model

  @required_fields ~w(user_id geodata)a
  @optional_fields ~w()a

  schema "geolocations" do
    field :user_id, :integer
    field :geodata, :map

    timestamps()
  end

  @doc """
  Builds a changeset based on the `struct` and `params`.
  """
  def changeset(struct, params \\ %{}) do
    struct
    |> cast(params, @required_fields ++ @optional_fields)
    |> validate_required(@required_fields)
  end
end
```

Moodul *Integration.Geolocation* on mudel, mis defineerib andmete jaoks vajaliku struktuuri. Antud juhul sisaldab *%Geolocation{}* struktuur kahte nõutud välja - *user_id*, mis on täisarv ning *geodata*, mis on sõnastik. Valikulisi välju hetkel ei ole, mistõttu on list *@optional_fields* tühi. Mudel sisaldab endas funktsiooni *changeset/2*, mis aitab teha andmetega vajalikud validatsioonid ja filtreerimised enne kui need andmebaasi salvestatakse.

Funktsioon *Repo.insert/1* saab argumendiks *changeset*'i ning kui viimane on valiidne, salvestatakse andmed andmebaasi. Kui *changeset* sisaldab infot, mis takistab andmete salvestamist, siis andmed andmebaasi ei jõua. Kontrolleri meetod tagastab lõpuks päringu tegijale vastuse, kas andmed salvestati või mitte.

3.3 Testimine

Tarkvaraarenduses on heaks tavaks koodi testida. See kasvatab rakenduse töökindlust ja annab arendajatele tagasisidet, kas allolev kood töötab nii nagu kavatsetud. Samuti on testitud koodi parem hallata ning tagantjärele paremaks muuta või uusi funktsionaalsusi lisada. Elixiri sisaldab *ExUnit* teeki, mis võimaldab lihtsalt teste kirjutada. See sisaldab endas vajalikku funktsionaalsust testimaks Elixiri programme.

Kõige lihtsamad testid on *Geolocation changeset*'i testimiseks. Need testid vaatavad, kas *changeset* on erinevate parameetrite korral valiidne.

```
defmodule Integration.GeolocationTest do
  use Integration.ModelCase

  alias Integration.Geolocation

  @valid_attrs %{geodata: %{}, user_id: 1}
  @invalid_attrs %{}

  test "changeset with valid attributes" do
    changeset = Geolocation.changeset(%Geolocation{},
@valid_attrs)
    assert changeset.valid?
  end

  test "changeset with invalid attributes" do
    changeset = Geolocation.changeset(%Geolocation{},
@invalid_attrs)
    refute changeset.valid?
  end
end
```

ExUnit *test* makro defineerib testi üksuse ning *assert/1* ja *refute/1* vaatavad, kas väärtus on tõene või väär.

Autentimis pistiku puhul peaks testima nii *validate_user_credentials/1* kui ka *call/2* funktsioone. Kuna mõlema funktsiooni tulemus sõltub põhirakenduse olekust, siis tuleb põhirakendusest tulev vastust teeselda.

```elixir
defmodule Integration.CoreAPIMock do
  def validate_user("123456") do
    {:ok, %{"success" => true, "user_id" => 1}}
  end
  def validate_user("invalid") do
    {:ok, %{"success" => false, "message" => "Unauthorized"}}
  end
end
```

Moodul *Integration.CoreAPIMock* sisaldab ühte funktsiooni kahe erineva signatuuriga. Esimesel juhul teeseldakse õnnestunud päringut põhirakendusse, teisel juhul ebaõnnestunud. Nüüd on võimalik testida autentimis pistikut muretsemata, et testide tulemused sõltuksid välistest teguritest.

```elixir
defmodule Integration.AuthTest do
  use Integration.ConnCase
  alias Integration.Auth

  describe "validate_user_credentials" do
    test "validate_user_credentials returns user_id if credentials
valid" do
      {:ok, user_id} = Auth.validate_user_credentials("123456")
      assert user_id == 1
    end

    test "validate_user_credentials returns error tuple when
credentials invalid" do
      assert {:error, "Unauthorized"} =
Auth.validate_user_credentials("invalid")
    end
  end
end
```

Funktsiooni *validate_user_credentials/1* puhul testitakse kahte juhtumit. Esimesel juhul on võti valiidne ning kontrollitakse, et funktsioon tagastaks enniku, mis sisaldab õiget kasutaja id'd. Teisel juhul on võti mittevaliidne ning kontrollitakse, et funktsioon tagastaks õiges formaatis enniku.

```
defmodule Integration.AuthTest do

  …

  describe "call" do
    test "call assigns user_id if credentials valid", %{conn:
conn} do
      conn =
        conn
        |> Map.put(:params, %{"user_credentials" => "123456",
"lastPosition" => "{}"})
        |> Auth.call(%{})
      assert conn.assigns[:user_id] == 1
    end

    test "halts conn if credentials invalid", %{conn: conn} do
      conn =
        conn
        |> Map.put(:params, %{"user_credentials" => "invalid",
"lastPosition" => "{}"})
        |> Auth.call(%{})
      assert conn.halted
    end
  end
end
```

Funktsiooni`call/2` puhul vaadatakse samuti kahte juhtumit. Kui `%Plug.Conn{}` struktuur sisaldab valiidseid parameetreid, siis salvestatakse `user_id` `conn.assigns` sõnastikku. Vastasel juhul vaadatakse, et edasine `conn`'i transformeerimine oleks peatatud.

Kontrollerite testimine on mõningal määral keerulisem, kuna erinevad rakenduse osad peavad tegema koostööd, kuid mõte jääb samaks. Testida oleks vaja edukat ja mitteedukat juhtu. Edukal juhul salvestatakse andmed baasi ning mitteeduka puhul mitte. Samuti saadetakse õige vastus päringu tegijale. Tänu `%Plug.Conn{}` ülesehitusele ning tõsiasjale, et kogu päringu ja vastuse informatsioon salvestakse ühte struktuuri, on ka kontrolleri meetodide testimine küllaltki lihtne.

```
defmodule Integration.GeodataControllerTest do
  use Integration.ConnCase

  alias Integration.Repo
  alias Integration.Geolocation

  @valid_attrs %{lastPosition: "{}"}

  test 'saves geodata when valid attributes', %{conn: conn} do
    conn =
      conn
      |> assign(:user_id, 1)
      |> post(geodata_path(conn, :geodata), @valid_attrs)

    assert conn.status == 200
    assert conn.resp_body == "Geolocation saved."
    assert Repo.one(from g in Geolocation, order_by: [desc: g.id],
limit: 1).user_id == 1
  end

  test 'does not save geodata when user not assigned', %{conn:
conn} do
    conn =
      conn
      |> post(geodata_path(conn, :geodata), @valid_attrs)

    assert conn.status == 300
    assert conn.resp_body == "Geolocation not saved."
    assert Repo.one(from g in Geolocation, order_by: [desc: g.id],
limit: 1) == nil
  end
end
```

Igas testi üksuses on vaja ligipääsu *conn*-ile. Seda aitab saavutada direktiiv *use Integration.ConnCase*, mis seab igale testi üksusele täiesti algelise ja põhimõtteliselt tühja *conn*-i. Erinevate testide korral saab *conn* struktuuri vajalikku olekusse viia ning siis valideerida tulemust.

Esimeses üksuses seatakse *conn.assigns* sõnastikku kasutaja *id* ning seejärel tehakse päring marsruudile, mis salvestab andmed. Kuna *conn* sisaldas kasutaja *id*-d, siis kontrollitakse, et vastuse staatuskood ja keha oleksid vastavad. Samuti kontrollitakse, kas andmed salvestati andmebaasi pärides *geolocations* tabelist viimase kirje ning vaadatakse, kas kirje *user_id* väli on sama, mis *conn*-s olev kasutaja *id*. Teise üksuse puhul kasutaja *id*-d ei määrata ning kontrollitakse samuti vastuse staatuskoodi ja keha. Samuti tehakse kindlaks, et uut kirjet andmebaasi ei lisata.

Teste saab jooksutada käsurealt käsuga *mix test*.

4. Reaalaja tugi

4.1 Protsessid

Erlangi virtuaalmasina protsessid on Elixiri peamised ehitusblokid. Sarnaselt objekt-orienteeritud keeltele, kus maailma modelleeritakse objektide abil, kasutatakse Elixiris selleks protsesse. Kuna tegu on virtuaalmasina protsessidega, siis on need väga kergekaalulised ja üksteise suhtes isoleeritud, mis tähendab seda, et nad ei jaga omavahel mälu. See teeb neist suurepärased tööriistad, mille abil lahendada paralleelsust nõudvaid probleeme. Protsessid suhtlevad oma vahel sõnumite abil. Igal protsessil on oma *PID* ehk identifitseerimistunnus. Teades viimast saab protsessile saata sõnumeid. Protsess töötleb igat sõnumit saabumise järjekorras. Üheks protsesside kasutuskohaks on Phoenixi kanalid. Kanalite abil on võimalik lisada lihtse vaevaga enda rakendusse reaalaja tugi. Nii nagu protsessidki põhineb kanalite töö sõnumite saatmises ja vastu võtmises.

4.2 Integratsiooni rakenduse reaalaja tugi

Reaalaja toe eesmärgiks on kasutajate GPS-andmete reaalajas jälgimine. See nõuab integratsiooni rakenduselt, et iga uue andmepaketti saabumise puhul edastatakse viimased õigesse kanalisse.

Rakendust genereerides genereeriti ka *web/channels/user_socket.ex* fail, mis sisaldab endas serveri poolset *WebSocket*-i konfiguratsiooni. Siin defineeritakse kanali nimi koos vastava mooduliga ning transpordi protokolliga.

```
defmodule Integration.UserSocket do
  use Phoenix.Socket

  channel "tracking:*", Integration.TrackingChannel

  transport :websocket, Phoenix.Transports.WebSocket

  …
end
```

Kanalid protsessivad erinevaid sündmuseid. Iga kanal peab implementeerima vähemalt ühe järgmistest meetoditest - *join/3*, *terminate/2*, *handle_in/3*, *handle_out/3*. Integratsiooni rakenduse puhul on tarvis implementeerida ainult *join/3*, mille abil luuakse ühendus.

```
defmodule Integration.TrackingChannel do
  use Phoenix.Channel

  def join("tracking:" <> company_id, _payload, socket) do
    {:ok, assign(socket, :company_id,
String.to_integer(company_id))}
  end
end
```

Ühendus luuakse vastavalt kanali nime sobitamisega. Kuna kasutajate jälgimine peaks toimuma ettevõtte põhiselt peab integratsiooni rakendus arvet pidama ka põhirakendusest saabuva ettevõtte identifitseerimistunnuse üle. Selleks tuleb *Integration.Auth* pistiku *call/2* meetodis salvestada lisaks kasutaja *id*-le ka ettevõtte *id*. Ettevõtte *id*-d kasutatakse hiljem andmete edastamiseks aktiivsetele kanalitele. Samuti sisaldab kanali nimi ettevõtte *id*-d.

```
defmodule Integration.Auth do
  …

  def call(%Plug.Conn{params: %{"user_credentials" =>
user_credentials}} = conn, _opts) do
    case validate_user_credentials(user_credentials) do
      {:ok, user_id, company_id} ->
        conn
        |> assign(:user_id, user_id)
        |> assign(:company_id, company_id)
      {:error, reason} ->
        conn |> put_resp_content_type("application/json") |>
send_resp(401, reason) |> halt
    end
  end

  …
end
```

Samuti muutus sellega põhirakendusest tuleva vastuse sisu, kuna nüüd saadetakse vastusena ka ettevõtte *id*. Seetõttu peab uue formaadiga arvestama ka *validate_user_credentials/1* meetod.

```
def validate_user_credentials(user_credentials) do
    case @core_api.validate_user(user_credentials) do
      {:ok, %{"success" => true, "user_id" => user_id,
"company_id" => company_id}} ->
        {:ok, user_id, company_id}

      {:ok, %{"success" => false, "message" => message}} ->
        {:error, message}

      {:error, reason} ->
        {:error, reason}
    end
  end
```

Serveri poole pealt on jäänud veel GPS-andmete edastamine lahtiolevatele ühendustele. Kuna andmete edastamine on mõtekas ainult juhul kui need on valiidsed, siis on mõistlik need edastada õnnestunud salvestamise korral. Selleks tuleb edastuse loogika panna kontrollerisse.

```
defmodule Integration.GeodataController do
  …

  def geodata(conn, %{"lastPosition" => last_position}) do
    data = %{"geodata" => Poison.decode!(last_position), "user_id"
=> conn.assigns[:user_id]}
    changeset = Geolocation.changeset(
      %Geolocation{},
      data
    )
    case Repo.insert(changeset) do
      {:ok, _geolocation} ->
        if conn.assigns[:company_id], do:
Integration.Endpoint.broadcast("tracking:#{conn.assigns[:company_i
d]}", "new_geodata", data)
        conn
        |> send_resp(200, "Geolocation saved.")
      {:error, _changeset} ->
        conn
        |> send_resp(300, "Geolocation not saved.")
    end
  end
end
```

Õnnestunud salvestamise puhul kontrollitakse, kas ettevõtte *id* on olemas ning saadetakse kõikidele lahtiolevatele kanalitele, mille nimi sisaldab vastavat ettevõtte *id*-d, teade *new_geodata* ning GPS-andmed.

5. Kokkuvõte

Käesoleva bakalaureusetöös tutvustati Elixiri programmeerimiskeele põhitõdesi ning keelt ümbritsevaid tööriistu ja raamistikke. Tutvustati Elixir andmetüüpe, andmestruktuure, funktsioone, kontrollstruktuure, mustrite sobitamist ja toruoperaatorit. Praktilise poole pealt loodi E-agronomi integratsiooni rakendus, mis salvestab kasutajate mobiilirakendusest tulevaid GPS-andmeid. Selle käigus vaadati, kuidas programm suhtleb teiste rakendustega, et autentida sissetulevad päringud ning kuidas Phoenixi raamistik suhtleb andmebaasiga. Neljandas peatükis pandi alus reaalaja toele, mis võimaldab kasutajaid reaalajas jälgida. Tänu Elixiri kompaktsusele ja erinevatele abstraktsioonidele sai integratsiooni rakenduse kood peaaegu täielikult välja toodud. Lisaks sai väljatoodud ka kogu rakenduse testkood. Edasine arendus eeldab JavaScripti kliendi kirjutamist, et GPS-andmed kaardil kuvada. Käesolevat tööd võib käsitleda ka kui esimest eestikeelset õppematerjali Elixiri programmeerimiskeele jaoks.

6. Viited

[1] Official Erlang website. https://www.erlang.org/ (09.05.2017)
[2] Jurič, S., Elixir in Action, Manning Publications, 2015
[3] Sahu, N. (2015), An Interview with Elixir Creator José Valim,
https://www.sitepoint.com/an-interview-with-elixir-creator-jose-valim/, (09.05.2017)
[4] Thomas, D., Programming Elixir 1.3, The Pragmatic Bookself, 2016
[6] Official Phoenix website. http://www.phoenixframework.org/ (09.05.2017)
[7] Plataformatec. (2012-2017). Official Elixir guide. Alias, require, and import. http://elixir-lang.org/getting-started/alias-require-and-import.html, (09.05.2017)

Lisad

I. Mõisted

Põhirakendus - Antud töö kontekstis on **põhirakendus** Eagronomi peamine toode ehk veebirakendus, mis võimaldab põllumehel mugavalt oma ettevõtte juhtimist planeerida. Põhirakendus on kirjutatud Ruby on Rails veebiraamistikuga.

Mobiilirakendus - Eagronomi **mobiilirakendus**, mille abil ettevõtte töötaja saab ennast kaardil positsioneerida ning varem planeerituid töid tehtuks märkida. Mobiilirakendus on kirjutatud React Native abil ning töötab nii Androidi kui ka iOS platvormil.

Integratsiooni rakendus - Töö käigus implementeeritud rakendus, mis suhtleb põhi- ja mobiilirakendusega ning salvestab mobiilirakendusest saadetavaid GPS-andmeid. Integratsiooni rakendus on kirjutatud Phoenixi veebiraamistikuga.

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Kaarel Tinn**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose **Veebirakenduse loomine funktsionaalses programmeerimiskeeles Elixir**, mille juhendaja on Kalmer Apinis,

    1. reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
    2. üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.
3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **11/05/2017**