Izrada web aplikacija pomoću okruženja Phoenix i programskog jezika Elixir

Teskera, Stjepan

Izrada web aplikacija pomoću okruženja Phoenix i programskog jezika Elixir

Teskera, Stjepan

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Stjepan Teskera

IZRADA WEB APLIKACIJA POMOĆU OKRUŽENJA PHOENIX I PROGRAMSKOG JEZIKA ELIXIR

Diplomski rad

Voditelj rada:
prof. dr. sc. Ivica Nakić

Zagreb, rujan, 2020.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Elixir je funkcijski programski jezik koji se pojavio 2011. godine. Temelj mu je Erlang, funkcijski jezik koji postoji od 1986., poznat po efikasnom sustavu izvođenja. Nastao je kao plod pokušaja da se poveća produktivnost Erlang virtualnog stroja BEAM. Jezik je visoke paralelnosti i niske latentnosti. Dizajniran je kako bi rukovao velikim količinama podataka. Njegova brzina i sposobnosti pomažu mu da se širi u telekomunikacijama i financijskoj industriji. U Elixiru je napisano okruženje Phoenix za razvoj web aplikacija i slijedi arhitekturalni obrazac MVC. Phoenix je nastao u svrhu pružanja vrlo učinkovitih aplikacija. Zanimljivo svojstvo koje pruža su kanali koji vanjskim klijentima omogućuju komunikaciju u stvarnom vremenu.

U prvom poglavlju opisujemo osnove jezika Elixir. Najprije objašnjavamo kako ga instalirati i koristiti. Govorimo više o tome što znači da je Elixir funkcijski jezik. Objašnjavamo kako se koriste varijable i funkcije. U svakom dijelu dajemo primjere kako bi se lakše shvatilo što se i kako događa. Zatim govorimo o kontroli toka i kako ju ostvariti u funkcijskom jeziku. Objašnjavamo što su to rekurzivne funkcije, čemu služe i kako ih pisati. Spominjemo i funkcije višeg reda i koja je njihova uloga. Na kraju poglavlja ćemo reći nešto više o nečistim funkcijama.

U drugom poglavlju opisujemo okruženje Phoenix. Za potrebe ovog rada izrađena je web-aplikacija za igranje dame čiju izgradnju pratimo od početka. Prvo objašnjavamo kako instalirati Phoenix i početi s njegovim korištenjem. Zatim objašnjavamo njegove osnovne dijelove poput upravljača i pogleda i kako oni nastaju. Dalje prelazimo na priču o Ectu i kako s njime upravljati bazom podataka. Zatim prelazimo na autentifikaciju i kako ju iskoristiti za prijavu korisnika. Na kraju ćemo reći više o kanalima i komunikaciji preko njih.

Poglavlje 1

Programski jezik Elixir

Elixir je dinamički, funkcijski programski jezik koji je pogodan za izradu skalabilnih i održivih aplikacija. Za pokretanje koristi Erlang VM, poznatu po niskoj latentnosti i distribuiranim sustavima otpornim na greške. Koristi se uspješno i u web razvoju, ugrađenim sustavima i domenama koje procesiraju multimedijske podatke. Zanimljivo svojstvo je da je u Elixiru sve što postoji izraz. Neki od najpoznatijih Elixir projekata su Mix, Phoenix i Ecto. Primjeri jednostavnijih Elixir aplikacija nalaze se u [12] i [6].

1.1 Instalacija i korištenje

Za korištenje Elixira potreban je Erlang, no Elixirov čarobnjak instalira i sam Erlang. Praćenjem uputa na službenoj stranici Elixira [3], Elixir se instalira lagano na bilo kojem operativnom sustavu. Nakon završetka instalacije, može se provjeriti uspješnost instalacije i verzija Elixira upisom naredbe u terminal: `elixir -v`. Glavni alat koji ćemo koristiti je Elixirova interaktivna ljuska, IEx, koju se može pokrenuti upisom naredbe `iex` u terminal. Ljuska je iznimno korisna za brzo isprobavanje koda i debuggiranje. U ljusci se mogu pokretati linije koda poput `IO.puts("Hello, World")`. Za prekid rada u ljusci potrebno je dva puta stisnuti `Ctrl+C`. Elixirove datoteke imaju ekstenziju .ex (kompajlirane datoteke) ili .exs (skriptne datoteke). Kod unutar kompajlirane datoteke kompajlira se u ljusci naredbom `c("Hello.ex")`, dok je kod unutar skriptne datoteke moguće izvršiti naredbom u terminalu `elixir helloworld.exs`.

1.2 Funkcijski jezik

U funkcijskoj programskoj paradigmi, funkcije su osnovni građevni blokovi, sve vrijednosti su nepromjenjive, a kod je deklarativan. Elixir je dinamički funkcijski jezik što znači

da za vrijeme izvođenja izvršava mnoge uobičajene programske zadatke koje statički jezici izvršavaju tijekom kompajliranja. Jedan od tih zadataka bi bio proširenje programa dodavanjem novog koda. Jednostavna sintaksa čini ga pristupačnim programskim jezikom za sve, bili oni upoznati s funkcijskom paradigmom ili ne.

Konvencionalni jezici koriste zajedničke promjenjive vrijednosti koje zahtijevaju dretve i mehanizme zaključavanja za istovremeni i usporedni rad. U funkcijskom programiranju, sve vrijednosti su nepromjenjive i time se miče potreba za mehanizmima zaključavanja što omogućuje jednostavniji usporedni rad. Nepromjenjivost podataka vidi se na sljedećem primjeru.

```
iex> list = [1, 2, 3]
[1, 2, 3]
iex> List.delete_at(list, 1)
[1, 3]
iex> list ++ [0]
[1, 2, 3, 0]
iex> list
[1, 2, 3]
```

Vrijednost liste je nepromjenjiva. Koja se god operacija primijeni na listi, stvaraju se nove vrijednosti. Ako je lista nepromjenjiva i svaka operacija daje sigurnu vrijednost, kompajler može sigurno izvršiti ove tri linije usporedno bez utjecaja na konačni rezultat.

U funkcijskom programiranju, funkcije su osnovni alati u građenju programa. Funkcije primaju podatke, obavljaju određene operacije i vraćaju vrijednost. Uglavnom su kratke i izražajne. Spajanjem više manjih funkcija stvara se veći program. Složenost građenja aplikacije smanjuje se kad funkcije imaju sljedeća svojstva: nepromjenjive vrijednosti, na rezultat funkcije utječu isključivo samo argumenti funkcije, funkcija nema nikakvog utjecaja osim vrijednosti koju vraća. Funkcije koje zadovoljavaju ta tri svojstva nazivaju se čiste funkcije. Primjer je funkcija koja primljeni broj povećava za 1.

U funkcijskom programiranju vrijednosti se uvijek eksplicitno šalju između funkcija, čime se jasno vidi što su ulazni, a što izlazni podaci. Kod objektno orijentiranih jezika, objekti imaju svoje stanje i metode koje mijenjaju to stanje. Funkcije se mogu koristiti i kao argumenti u drugim funkcijama.

Elixir je fokusiran na tok pretvorbe podataka, ima poseban operator `|` zvan `pipe` operator kojim se može spajati više funkcijskih poziva i rezultata. Koristeći `pipe` operator, rezultat svakog izraza šalje se sljedećoj funkciji.

```
iex> [1, 2, 3] |> List.delete_at(1) |> List.insert_at(1, 0)
[1, 0, 3]
```

Imperativno programiranje fokusira se na ”kako riješiti problem”, opisujući svaki korak. Funkcijsko programiranje je deklarativno. Deklarativno programiranje fokusira se na ”što je nužno za riješiti problem”, opisujući tok podataka.

1.3 Varijable i funkcije

Varijable i funkcije su osnova svakog funkcijskog jezika. U Elixiru su čak i funkcije vrijednosti. Vrijednosti su sve što može predstavljati podatke u Elixiru. Literali prestavljaju vrijednosti koje ljudi mogu lagano razumjeti. Elixir se sam brine da literale pretvori u format za strojeve.

Tip	**Primjena**	**Primjer**
integer	cijeli brojevi	3, 5
float	realni brojevi	3.14, 0.5
string	tekst	”Hello, World!”, ”elixir”
boolean	logičke operacije	true, false
atom	identifikatori	:ok, :error
tuple	kolekcije poznate veličine	{1, 2, 3}, {:ok, ”elixir”}
list	kolekcije nepoznate veličine	[1, 2, 3], [”a”, ”b”, ”c”, ”d”]
map	traženje vrijednosti u rječniku po ključu	%{id: 1, name: ”aa”}, %{1 => ”a”}

Tablica 1.1: Neki od tipova u Elixiru

Tip atom je konstanta čije ime je i njezina vrijednost. True, false i nil su atomi :true, :false i :nil.

Elixir može proizvesti rezultat za bilo koji izraz. Računalu dajemo izraze, a ono nam vraća rezultat. Najjednostavniji izraz je vrijednost.

```
iex> 14
14
iex> 1 + 2
3
```

Brojevi 1 i 2 su vrijednosti a + je operator. Operatori računaju vrijednosti i proizvode rezultat. Možemo slagati više operatora i vrijednosti. Svaki operator izvršava se u određenom redoslijedu, ima svoj prioritet. Na primjer, * se izvršava prije +, ali ako dodamo zagrade, prije se računa izraz u zagradama.

```
iex> 1 + 2 * 3
7
```

```
iex> (1 + 2 ) * 3
9
```

Operator	**Primjena**	**Primjer**
+	zbrajanje brojeva	1 + 1, 0.5 + 0.5
-	oduzimanje brojeva	1 - 1, 2.5 - 1.2
*	množenje brojeva	1 * 1, 1.5 * 2.3
/	dijeljenje brojeva	1 / 1, 2.8 / 1.5
==	provjera jesu li dvije vrijednosti jednake	1 == 1.0, 1 == 2
!=	provjera jesu li dvije vrijednosti različite	1 != 1.0, 1 != 2
<	provjera je li lijeva vrijednost manja od desne	1 <2, 2 <1
>	provjera je li lijeva vrijednost veća od desne	2 >1, 1 >2
<>	konkateniranje dva stringa ili binaryja	”Hello, ” <>”World”
++	konkateniranje dvije liste	[0, 1] ++ [2, 3]

Tablica 1.2: Neki od operatora u Elixiru

Logički izrazi koriste se uglavnom za stvaranje uvjeta koji kontroliraju tok programa. U Elixiru logički operatori imaju dvije verzije. Logički operator *ILI* ima `or` i `||`, *I* ima `and` i `&&`, a *NE* ima `not` i `!`. Operatori `or` , `and` i `not` očekuju boolean kao prvi argument, a u protivnom se dobija greška. Operatori `||`, `&&` i `!` mogu s lijeve strane primiti istinite (eng. *truthy*) i lažne (eng. *falsy*) vrijednosti. Lažne vrijednosti su `false` i `nil`, a istinite sve vrijednosti koje nisu lažne.

Varijabla sadrži vrijednost. Vrijednost se varijabli dodjeljuje operatorom =. Konvencija u Elixiru je da se imena varijabli pišu u *snake_case* formatu, malim slovima i s _ između riječi.

Funkcije primaju ulazne podatke, obavljaju neke operacije i vraćaju izlaznu vrijednost. U tijelu funkcije piše se što funkcija treba raditi, koje operacije treba obavljati. Zadnja vrijednost u tijelu funkcije vraća se kao izlazna vrijednost funkcije.

```
iex> add = fn addend1, addend2 -> addend1 + addend2 end
iex> add.(1,2)
3
```

Napravili smo *anonimnu funkciju* (također znanu i kao *lambda funkciju*) i pridružili ju varijabli `add`. Anonimne funkcije pozivaju se varijablom kojoj su dodijeljene iza koje slijedi `.` i u zagradama vrijednosti koje šaljemo funkciji odvojene zarezima. Definiciju funkcije započinjemo s `fn`. Zatim slijede argumenti odvojeni zarezima. Nakon argumenata, operator `->` označava da je dalje tijelo funkcije. U tijelu funkcije pišu se izrazi i vraća se vrijednost zadnjeg izraza. Definicija funkcije završava s `end`.

Funkcije su vrijednosti tipa `function`. Posljedica toga je da funkcije mogu primati druge funkcije kao argument.

```
iex> increase_once = fn addend1, addend2, increase ->
increase.(addend1 + addend2) end
iex> increase = fn val -> val + 1 end
iex> increase_once.(5, 6, increase)
12
```

Djelokrug (eng. *scope*) je dio programa koji se odnosi na vidljivost varijabli u kodu. Funkcijin djelokrug je sve što je definirano u funkciji. Ako se u funkciji koristi varijabla koja nije definirana u njenom djelokrugu, kompajler traži tu varijablu u kodu prije i izvan funkcije, te dolazi do greške ako ta varijabla ni tamo nije definirana. Kad definiramo funkciju koja koristi varijablu van njezinog djelokruga, uzme se trenutna vrijednost te varijable i ona je nepromjenjiva. Ako se izvan funkcije kasnije u kodu vrijednost te varijable promijeni, naša funkcija će i dalje koristiti staru vrijednost.

Imenovane funkcije definiraju se unutar modula u Elixiru. Funkcija iz modula poziva se upisom imena modula i imena funkcije s `.` između. Primjer poziva funkcije iz modula `String` je `String.Upcase("string")`. Elixir sam pruža mnogo korisnih modula s definiranim raznovrsnim funkcijama. To je sve dokumentirano u Elixirovoj službenoj dokumentaciji [4]. `Kernel` je Elixirov modul čije su funkcije dostupne bez upisa imena modula. Taj modul sadrži osnovne stvari kao što su aritmetičke operacije te makroi za kontrolu toka i definiranje novih funkcionalnosti.

Vlastiti modul može se lagano izraditi. Definira se s `defmodule` čime započinje definicija modula. Zatim slijedi ime modula i `do`, čime započinje tijelo modula gdje se definiraju, uvoze i pozivaju funkcije, a definicija modula završava s `end`. Imena modula pišu se u u *CamelCase* formatu, velikim početnim slovima riječi, a riječi se pišu zajedno. Ime datoteke koja sadrži definiciju nekog modula je isto kao i samo ime modula, ali piše se malim slovima, u *snake_case* formatu i s `.ex` na kraju što označava da Elixir datoteku koja se treba kompajlirati.

Funkcije unutar modula započinju definiciju s `def` nakon čega slijedi ime funkcije. Konvencija je da se imena funkcije isto pišu u *snake_case* formatu kao i varijable. Zatim se u zagradama navode argumenti odvojeni zarezima. Slijedi `do` čime započinje tijelo funkcije. U tijelo se upisuju izrazi kojima se postiže što funkcija treba napraviti te se vraća vrijednost zadnjeg izraza u tijelu funkcije. Definicija funkcije završava s `end`.

Konvencija je da se u aplikacijama, aplikacijskim modulima ispred imena samog modula nadoda ime aplikacije i `.` kako bi se izbjeglo miješanje s Elixirovim modulima. Primjer je `MyApp.MyModule` umjesto samo `MyModule`.

Stvorene imenovane funkcije koriste se kao Elixirove pružene funkcije. Korištenjem `import` direktive, funkcije iz uveženog modula funkcioniraju kao funkcije `Kernel` mo-

dula, ne mora se navoditi ime modula prije imena funkcije koja se koristi. Primjer je `import String, only: [split:3]`. Kasnije se onda u kodu umjesto `String.split` može pisati samo `split`. U `import` direktivi nakon imena funkcije dolazi broj koji se zove *mjesnost funkcije*. Taj broj označava koliko argumenata funkcija prima. U Elixirovoj dokumentaciji, funkcije su označene s `ime_funkcije/mjesnost`. U dokumentaciji, funkcija iz primjera `String.split` ima dvije definicije, jedna je mjesnosti 1 (označena s String.split/1), a druga je mjesnosti 3 (označena s String.split/3). Korištenjem opcije `only` uvoze se samo određene funkcije, no `only` se može i izostaviti te se onda uvoze sve funkcije iz modula.

Imenovane funkcije ne mogu se samo pridijeliti varijablama, ali postoje načini na koje možemo postići takvu funkcionalnost.

```
iex> split = fn string -> String.split(string) end
iex>split.("hello world")
["hello", "world"]
iex> split = &String.split/1
iex> split.("hello world")
["hello", "world"]
```

Za korištenje funkcije `String.split/1` kao varijable može ju se omotati u anonimnu funkciju proslijeđujući zadani argument ili se pomoću operatora & može uhvatiti referencu na funkciju. Operatorom & mogu se i stvoriti anonimne funkcije. U tom slučaju & označava početak funkcije, nakon čega u zagradama piše tijelo funkcije. `&1` i `&2` predstavljaju prvi i drugi argument koje funkcija prima. U sljedećem primjeru je anonimna funkcija stvorena pomoću & koja zbraja dva broja.

```
iex> add = &(&1 + &2)
iex> add.(1,2)
iex> 3
```

1.4 Kontrola toka programa

Za prepoznavanje podudarajućih uzoraka (eng. *pattern matching*) koristi se operator =. Ako se uzorci s obje strane operatora = ne podudaraju, nastaje `MatchError` i zaustavlja se program, a u protivnom se program nastavlja izvršavati.

```
iex> 0 = 0
iex> 0 = 1
iex> x = 0
iex> 0 = x
```

```
iex> 1 = x
iex> x = 1
iex> ^x = 1
iex> ^x = 0
```

U prvoj liniji vrijednosti se podudaraju, u drugoj dolazi do greške. U trećoj liniji Elixir pridružuje varijabli `x` vrijednost 0 pa se vrijednosti podudaraju. U čevrtoj liniji vrijednosti se opet podudaraju jer je prethodno varijabli `x` pridružena vrijednost 0. U petoj liniji vrijednosti se ne podudaraju jer je vrijednost varijable `x` 0. U šestoj liniji dodijeli se nova vrijednost varijabli `x` i vrijednosti se podudaraju. *Pin* operator ^ služi da se izbjegne dodijeljivanje nove vrijednosti varijabli. U sedmoj liniji vrijednosti se podudaraju, ali u zadnjoj ne jer je vrijednost varijable `x` u tom trenutku 1.

Prepoznavanje podudarajućih uzoraka koristi se i u izvlačenju dijelova vrijednosti iz različitih tipova podataka kao što su dio stringa, element liste ili vrijednost mape. Dio stringa može se izvući na sljedeći način:

```
iex> "Full name: " <> name = "Full name: Stjepan Teskera"
iex> name
"Stjepan Teskera"
```

Varijabla se ne smije koristiti na lijevoj strani operatora <> jer je string binaran tip i <> binarni operator, a izraz u tom slučaju ne može započeti varijablom bez pružanja njezine binarne veličine.

Torke (eng. *tuples*) su kolekcije koje su u memoriji spremljene susjedno, čime se omogućuje brzo pristupanje elementima po indeksu. Torke su česta pojava u Elixiru i glavna uloga im je slanje signala zajedno s vrijednošću. Primjeri su {`:ok, 1`} ili {`:error, :not_found`}.

```
iex> {:ok, a, b, _} = {:ok, 0, 1 + 2, 4}
```

Elixir u ovom slučaju pridružuje više vrijednosti varijablama. Varijabli `a` pridružuje vrijednost 0, varijabli `b` pridružuje 3, a _ označava da se taj dio zanemaruje. Torke su korisne i za signaliziranje uspjeha ili neuspjeha u izlaznoj vrijednosti funkcije kada funkcija kao prvi element vraća `:ok` ili `:error`. Kao i inače, u slučaju nepodudaranja dolazi do greške.

Torke su u memoriji spremljene susjedno što znači da unaprijed moramo znati koliko elemenata imaju. Kada to predstavlja problem, Elixir ima rješenje u obliku liste. U Elixiru, liste su vezane liste, što znači da svaki element sadrži vrijednost i referencu na sljedeći element. Elixir za liste pruža poseban operator | koji odvaja početne elemente od ostatka liste.

```
iex> [x, x, y, _, y | rest] = [0, 0, "string", 2, "string", 3, "b"]
```

U ovom slučaju, prvih 5 elemenata odvojeno je od ostatka liste. Po uzorku se vidi da prva dva elementa moraju biti ista, treći i peti moraju biti isti, a četvrti se zanemaruje. Varijabli `x` pridružuje se vrijednost 0, varijabli `y` pridružuje se string `"string"`,dok se varijabli `rest` pridružuje ostatak liste. Ako ostatka liste nema, pridružuje se prazna lista, međutim operator `|` mora se koristiti na listama s barem jednim elementom.

Mapa je tip podataka na principu para ključ/vrijednost. Ključ i vrijednost mogu biti bilo koji tipovi, a ključ mora biti jedinstven. Mapa se može definirati na dva načina.

```
mapa = %{name: "Stjepan", surname: "Teskera", college: "
   PMF"}
mapa = %{:name => "Stjepan", :surname => "Teskera",
   college: "PMF"}
%{name: "Stjepan", surname: surname} = mapa
```

U prepoznavanju podudaranja mogu se provjeravati ključevi i vrijednosti, a ako se koristi prazna mapa, ona se podudara sa svim mapama. Istovremeno se mogu i izvlačiti vrijednosti i provjeravati. Ako mapa ne sadrži ključ koji provjeravamo, dolazi do greške. U prethodnom primjeru prvo je definirana mapa, a zatim se provjeravalo postoji li u njoj ključ `:name` s vrijednošću `"Stjepan"` i ako da, varijabli `surname` pridružila se vrijednost ključa `:surname`.

Lista ključnih riječi (eng. *keyword list*) je lista torki od 2 elementa i dopušta duplicirane ključeve, ali oni moraju biti atomi.

```
iex> [x, y] = [z: 0, z: 1]
```

Struct je ekstenzija mape. Struct se koristi za predstavljanje konzistentnih struktura koje imaju iste ključeve svugdje u aplikaciji. Provjera podudaranja je kao s mapama. Sigil je jednostavni tekstualni prečac za stvaranje vrijednosti. D je sigil za struct `Date` koji sadrži ključeve `year`, `month`, `day` i `calendar`, a koristi se u sljedećem primjeru.

```
date = ~D[2020-01-01]
%Date{year: year} = date
%Date{year: year} = %{year: 2020}
```

U drugoj liniji dolazi do podudaranja vrijednosti i varijabli `year` pridružuje se 2020. U trećoj liniji dolazi do greške jer mapa nije struct `Date` iako je svaki `Date` mapa.

Funkcije mogu imati zadane (eng. *default*) argumente koji se postavljaju operatorom `\\`. U tom slučaju Elixir stvara dvije verzije te funkcije. U sljedećem primjeru Elixir stvara funkcije mjesnosti 0 i 1, i kad ju pozivamo, Elixir se pobrine da pozove pravu s obzirom na to je li dan argument.

welcome.ex

```
defmodule Welcome do
  def hello(name \\ "you"), do: "hello " <> name
end
```

Kontrola toka može se obavljati funkcijama. Slijedi jedan takav primjer.

compare.ex

```
defmodule Compare do
  def less(number, another_number) do
    test(number <= another_number, number,
       another_number)
  end
  defp test(true, number, _), do: number
  defp test(false, _, another_number), do:
     another_number
end
```

U funkciji `less` gleda se je li prvi broj manji ili jednak drugome i boolean rezultat usporedbe prosljeđuje se funkciji `test` uz dane brojeve. Funkcija `test` će ovisno o rezultatu usporedbe brojeva vraćati manji broj, a definirana je pomoću `defp` čime se definiraju privatne funkcije modula, što znači da se one mogu koristiti isključivo unutar modula. Višestruke definicije funkcije (u ovom slučaju to je funkcija `test`) nazivaju se *funkcijske klauzule* (eng. *function clauses*) te one moraju biti definirane jedna za drugom, a kad se poziva funkcija Elixir će izvršiti funkciju prve klauzule koja odgovara.

Prethodni primjer može se poboljšati korištenjem *zaštitnih klauzula* (eng. *guard clauses*). Zaštitnim klauzulama možemo dodati boolean izraze u funkcije za uvjetovano izvršavanje. U zaštitnim klauzulama ne mogu se koristiti sve funkcije jer ta provjera mora biti iznimno brza i bez popratnih posljedica. Novi primjer ima manje teksta i čitljiviji je.

compare.ex

```
defmodule Compare do
  def less(number, another_number) when number <=
     another_number, do: number
  def less(_, another_number), do: another_number
end
```

Zaštitna klauzula u ovom slučaju je `when number < another_number`. Ako klauzula vraća `true`, izvršit će se ta funkcija, u suprotnom izvršit će se druga. Druga funkcija će se uvijek izvršiti ako se prva ne izvrši jer druga nema nikakvih uvjeta. Isti primjer može se napisati i anonimnom funkciju koja slijedi.

```
less = fn
  number, another_number when number <= another_number
     -> number
  _, another_number -> another_number
end
```

Elixir ima svoje ugrađene strukture za kontrolu toka poput `if`, `unless`, `cond` i `case`. Kad neki izraz rezultira istinito, od velike pomoći može biti kontrola `if`.

if_compare.ex

```
defmodule IfCompare do
  def equal(number, another_number) do
    if number == another_number do
      "equal"
    else
      "not equal"
    end
  end
end
```

Kad je izraz unutar kontrole `if` istinit, u ovom slučaju ispisat će se string `"equal"`, u suprotnom ispisat će se `"not equal"`. Postoji i komanda `unless`, koristi se na isti način kao i `if`, ali komandni blok za `unless` izvršava se kad je izraz lažan ili `nil`. U prošlom primjeru, kontrolu `if` moglo se pozvati i na sljedeći način: `if(number == another_number, do: "equal", else: "not equal")`

Kontrola `cond` koristi se kada se želi provjeriti različite varijable i vrijednosti u logičkim izrazima, a ne treba nam prepoznavanje podudaranja. U sljedećem primjeru koristi se `cond` struktura da bi se za uneseni broj dobio odgovor ima li 1, 2 ili više znamenaka.

cond_digits.ex

```
defmodule CondDigits do
  def digits (number) do
    result = cond do
      number < 10 -> "1 digit"
      number < 100 -> "2 digits"
      number >= 100 -> "3 or more digits"
    end
  end
end
```

Kontrolu toka može se postići naredbom `case`, kojom se može provjeravati više uvjeta. Unutar `case` kontrole, mogu se koristiti i zaštitne klauzule. Ako ne odgovara nijedan uvjet, doći će do greške. U primjeru koji slijedi koristi se `case` da bi se za unešeni string dobio odgovor može li se parsirati u pozitivan cijeli broj. Napomenimo da će zbog zaštitne klauzule doći do greške ako je unešen negativni broj.

case.ex

```
defmodule Case do
  def positive(input) do
    result = case Integer.parse(input) do
      :error ->
        "Must be integer"
      {x, _} when x > 0 ->
        "Is a positive integer"
      end
    IO.puts result
  end
end
```

1.5 Rekurzivne funkcije

Podaci su nepromjenjivi u funkcijskom programiranju te se ponavljanja ne mogu obavljati pomoću npr. `for` petlje. Za potrebna ponavljanja koriste se rekurzivne funkcije, funkcije koje pozivaju same sebe. Ograničena rekurzija je rekurzija čije će se izvršavanje zaustaviti nakon određenog broja poziva. Broj poziva koje će ograničena rekurzija obaviti ovisi o njenim argumentima. Razmotrimo sljedeću datoteku i pozovimo funkciju definiranu u njoj.

recursion.ex

```
defmodule Recursion do
  def print_multiple(string, n) when n <= 1, do: IO.puts
      string
  def print_multiple(string, n) do
    IO.puts string
    print_multiple(string, n-1)
  end
end
```

```
iex> c("recursion.ex")
iex> Recursion.print_multiple("abc", 5)
```

U modulu `Recursion` napravljene su dvije funkcije. Prva je granična klauzula koja nas ograničava da se rekurzija ne izvršava beskonačno i ona samo ispisuje zadani string. Granične klauzule uvijek moraju biti definirane prije ostalih klauzula. Druga funkcija ispisuje dani string i poziva samu sebe rekurzivno. U ovom primjeru, druga funkcija će se pozvati 4 puta, i onda će se pozvati prva funkcija čime će se izvršavanje završiti i rezultirati ispisom stringa ”abc” 5 puta.

U sljedećoj datoteci primjer je korištenja liste u rekurziji.

list_recursion.ex

```
defmodule ListRecursion do
  def data do
  [
    %{number: 5, multiplied: false},
    %{number: 6, multiplied: true},
    %{number: 8, multiplied: false}
  ]
  end
  def multiply([]), do: []
  def multiply([head = %{multiplied: true} | tail]) do
    [head | multiply(tail)]
  end
  def multiply([head | tail]) do
    new_head = %{number: head.number * 5, multiplied:
       true}
    [new_head | multiply(tail)]
  end
end
```

```
iex> c("list_recursion.ex")
iex> ListRecursion.multiply(ListRecursion.data)
```

Dani su podaci u obliku liste čiji su elementi mape. U svakoj mapi dan je broj i podatak je li taj broj već pomnožen s 5. Prva klauzula je granična, kada primi praznu listu završit će se izvršavanje. Treća klauzula uzima prvi element liste te broj unutar mape množi s 5 i varijablu `multiplied` postavlja na true te vraća novi podatak i poziva rekurziju s ostatkom liste. Druga klauzula provjerava je li broj u mapi već pomnožen, i ako jest, vraća mapu i

poziva rekurziju s ostatkom liste. Kada ne bi bilo druge klauzule, broj unutar mape množio bi se svaki put, iako je već varijablu `multiplied` imao postavljenu na true.

Neograničena rekurzija je rekurzija za koju ne možemo predvidjeti koliko će puta pozvati samu sebe. Primjer je pretraživanje i popisivanje direktorija na operativnom sustavu. Ulazi se u direktorij i zatim sve njegove poddirektorije, koji opet mogu sadržavati svoje poddirektorije. Da se izbjegnu razni problemi, rekurzija se može ograničiti, npr. vremenski ili dubinski, ali ni to ne garantira da neće biti duplikata. Problem nastaje kada u nekom direktoriju postoji prečac za drugi direktorij te u tom slučaju može doći do beskonačne petlje. U tom slučaju trebalo bi provjeriti je li direktorij pravi direktorij ili prečac te ako je prečac zaustaviti tu granu.

1.6 Funkcije višeg reda

Funkcije višeg reda su funkcije koje u argumentima imaju druge funkcije i/ili vraćaju funkcije. Omogućuju nam stvaranje funkcija pomoću jednostavnih sučelja. Primjer takve funkcije je funkcija `spawn`.

```
iex> spawn fn -> IO.puts("hello") end
```

`spawn` započinje proces i zove zadanu funkciju. U pozadini zapravo alocira memoriju i čini je dostupnom jezgri procesora. Prednost jednostavnog sučelja je da se brinemo jedino oko toga što ćemo dati procesu da radi. Slijedi primjer jedne funkcije višeg reda.

higher_order.ex

```
defmodule HigherOrder do
  def data do
  [
    %{number: 5, multiplied: false},
    %{number: 6, multiplied: true},
    %{number: 8, multiplied: false}
  ]
  end
  def each([], _function), do: nil
  def each([head | tail], function) do
    function.(head)
    each(tail, function)
  end
end
```

```
iex> c("higher_order.ex")
iex> HigherOrder.each(HigherOrder.data, fn item ->
IO.puts item.number end)
```

Funkcija `each` prima 2 argumenta, listu i funkciju. Za svaki element liste pozivat će se dana funkcija te će se funkcija rekurzivno zvati dok ne dođe do kraja liste. U ovom slučaju, za svaku mapu iz liste, ispisivat će se vrijednost ključa `number`.

Modul `Enum` nudi set algoritama za rad s `enumerable` tipom. U Elixiru, `enumerable` je bilo koji podatkovni tip koji implementira `Enumerable` protokol. Među njima su već prije spomenute liste, liste ključnih riječi i mape. U modulu se nalaze brojne korisne funkcije i lagano se koriste, među kojima je i `each` . Kompletan popis može se vidjeti u službenoj dokumentaciji Elixira.

Učestalo je prolaziti kroz `Enumerable`, filtrirati rezultate i mapirati vrijednosti. Tu nam pomažu komprehenzije koje grupiraju učestale zadatke u specijalnu formu `for`.

```
iex> for {number, fruit} <- [{1, "apple"}, {2, "banana"},
{3, "mango"}, {4, "pear"}, {5, "strawberry"}], number > 3, do: fruit
["pear", "strawberry"]
```

Izraz nakon `for` je generator. On generira vrijednosti koje će se koristiti u komprehenziji. Bilo koji `enumerable` može se proslijediti desnoj strani izraza generatora. Unutar komprehenzije može se i filtrirati. Također se može imati više generatora. U opciji `do` određuje se što će se nalaziti u novoj listi.

Pipe operator `|>` prosljeđuje izraz s lijeve strane operatora kao prvi argument funkcijskom pozivu s desne strane. Svrha mu je naglašavanje transformacije podataka nizom funkcija.

pipe.ex

```
defmodule Pipe do
  def list_to_capitalized_string(input) do
    input
      |> List.flatten()
      |> Enum.map(&String.capitalize/1)
      |> Enum.join(" ")
  end
end
```

```
iex> c("pipe.ex")
iex> Pipe.list_to_capitalized_string(["elixir", ["is", "a"],
["functional", "language"]])
"Elixir Is A Functional Language"
```

Funkcija prima listu listi stringova. Prvi korak je sravnjivanje te liste na samo listu stringova. Zatim se lista stringova prosljeđuje funkciji `Enum.map/2`, koja je mjesnosti 2, ali zbog pipe operatora koji je već proslijedio prvi argument, potrebno je još samo zadati drugi argument. To je upravo funkcija koja svakom stringu početno slovo postavlja na veliko. Zadnji korak funkcije je združivanje stringova u jedan, međusobno odvojeni razmacima. Pogledajmo kako bi funkcija mogla izgledati bez pipe operatora.

pipe.ex

```
defmodule Pipe do
  def list_to_capitalized_string(input) do
    Enum.join(
      Enum.map(
        List.flatten(input),
        &String.capitalize/1
      ), " "
    )
  end
end
```

Funkcija nije lako čitljiva, a da bi se pratilo što se događa u funkciji potrebno je čitati odozada. Suprotno tome, korištenjem pipe operatora, kod se čita lagano i otpočetka kako se izvršava i transformacija podataka. Kao što se vidi iz priloženog, pipe operator je iznimno koristan kad se zaredom zove više funkcija i čini korake transformacije podataka očitim.

Lijena evaluacija je strategija koja odgađa evaluaciju izraza dok njegova vrijednost nije potrebna. Razlog zašto postoji lijena evaluacija je ušteda na vremenu i memoriji, da bi se bitnije stvari izvršile ranije, a one manje potrebne u prigodnom trenutku. Funkcije višeg reda koriste se u kombinaciji s lijenom evaluacijom kada se radi s funkcijama koje se mogu izvršiti kasnije. `range` je primjer lijene kolekcije.

```
iex> 1..100
1..100
```

Kao što se vidi, `range` ne stavlja sve brojeve u memoriju. Broji od 1 do 100, ali ako ne pristupamo brojevima, zbog lijenosti evaluacije, nećemo vidjeti sve brojeve nego samo početni i zadnji, jer ostali nisu potrebni. Da bi se vidjeli svi brojevi, treba se napraviti neka operacija koja ih koristi.

```
iex> range = 1..5
iex> stream = Stream.map(range, &(&1 * 10))
iex> Enum.map(stream, &(&1 + 1))
[11, 21, 31, 41, 51]
```

`Stream` je modul koji sadrži funkcije za izradu i sastavljanje tokova. *Tok* (eng. *stream*) je lijeni enumerable. `range` je jedan primjer toka. Modul `Stream` omogućuje rad s enumerableima bez stvarnog prebrojavanja. U prošlom primjeru počinjemo kolekcijom `range` od 1 do 5. Zatim napravimo tok koji sve brojeve pomnoži s 10. U tom trenutku, nikakvo računanje se još nije dogodilo. Tek kad se pozove `Enum.map/2` se zapravo nabroje brojevi od 1 do 5, pomnože s 10 te im se doda 1. Vidljiva je razlika među funkcijama iz modula `Stream` i `Enum`. Funkcije iz modula `Stream` su lijene, dok su funkcije iz modula `Enum` željne (eng. *eager*).

1.7 Nečiste funkcije

Kako bi kod bio dobar, jedna od glavnih stvari na koju treba paziti je nepredvidljivost. Ako se neki dijelovi ponašaju nepredvidljivo, posao programera je da te dijelove učini predvidljivima. Čiste funkcije ponašaju se predvidljivo, dok se nečiste ponašaju nepredvidljivo. Čiste funkcije za dani ulaz uvijek vraćaju isti izlaz, bez obzira na broj pokušaja. Usto, nikad nemaju nikakvog drugog popratnog utjecaja, osim onog što trebaju napraviti.

```
iex> fnc = &(&1 + &2 * &3)
iex> fnc.(2, 5, 8)
42
iex> fnc.(1, 2, nil)
** (ArithmeticError)
```

Čiste funkcije mogu vraćati i greške, ali samo ako su i dalje predvidljive. Imaju svojstvo *referencijalne transparentnosti*, tj. cijeli poziv funkcije može se zamijeniti njenom odgovarajućom vrijednošću bez utjecaja na ostatak programa, npr. `fnc.(2, 5, 8)` može se zamijeniti s 42.

Nečiste funkcije za dani ulaz mogu vraćati različite izlaze, a mogu imati i popratne utjecaje van same funkcije, kao što su čitanje i pisanje datoteke ili traženje korisničkog unosa.

```
iex> DateTime.utc_now()
```

Funkciju `DateTime.utc_now()` uvijek se zove na isti način, bez argumenata. Funkcija vraća uvijek trenutno vrijeme, tako da će u 2 uzastopna poziva vratiti različite vrijednosti, što ju čini nečistom. Međutim nečiste funkcije katkada su nužnu, samo s njima treba pomno rukovati.

U borbi protiv nepredvidljivosti nečiste funkcije, od pomoći mogu biti strukture kontrole toka poput `case`. Slijedi primjer funkcije koja traži od korisnika unos 2 broja koja će zatim pomnožiti i ispisati.

impure.ex

```
defmodule Impure do
  def multiply() do
    input = IO.gets("First factor to multiply: ")
    result = case Integer.parse(input) do
      :error -> "Not a number"
      {number, _} ->
        another_input = IO.gets("Second factor to
           multiply: ")
        case Integer.parse(another_input) do
          :error -> "Not a number"
          {another_number, _} -> Integer.to_string(
             number * another_number)
        end
    end
    IO.puts(result)
    end
end
```

Funkcija prvo traži unos broja preko nečiste funkcije `IO.gets`. Ako se ne upiše broj, `Integer.parse/2` može proizvesti grešku. Zato se koristi `case` koji provjerava je li došlo do greške ili je unešen broj. Ako je došlo do greške, u varijablu `result` sprema se string ”Not a number”. Ako je unešen broj, traži se unos drugog broja koji će se pomnožiti s prvim. Opet se koristi `Integer.parse/2` u kombinaciji s `case` da se provjeri je li broj zapravo unešen te ako jest, u `result` se sprema umnožak dva broja. Na kraju se ispisuje vrijednost varijable `result`. Kao što se vidi, već za dvije strukture `case` kod izgleda pomalo nezgrapno tako da se treba pažljivo postupati.

Neke funkcije mogu proizvoditi greške ili izbacivati vrijednosti. U Elixiru je konvencija da se stavlja `!` na kraj imena takvih funkcija, npr. `DateTime.now!/2`. U takvim situacijama može pomoći struktura `try`. `try` omotava blok koda, i ako je u bloku došlo do greške, upotrebom komande `rescue` može se oporaviti. Do greške (ili iznimke) dolazi kad se u kodu događaju izvanredne stvari. Ako pak u bloku dođe do izbacivanja vrijednosti, ista se može uhvatiti komandom `catch`.

U Elixiru, funkcije podižu greške samo kad su u izvanredno pogrešnim situacijama. Slijedi primjer oporavka od greške.

impure.ex

```
defmodule Impure do
  def multiply() do
    try do
```

```
      input = IO.gets("First factor to multiply: ")
      {number, _} = Integer.parse(input)
      another_input = IO.gets("Second factor to multiply
         : ")
      {another_number, _} = Integer.parse(another_input)
      Integer.to_string(number * another_number)
    rescue
      MatchError -> "Not a number"
    end
  end
end
```

U `try` bloku su naredbe koje će se izvršiti u slučaju da ništa ne pođe po zlu. U `rescue` bloku su naredbe koje rukuju greškom. Ako korisnik ne upiše broj, varijabli `number` neće se uspjeti dodijeliti vrijednost i doći će do `MatchError` greške. U `rescue` bloku tražit će se `MatchError` uvjet i što u tom slučaju napraviti. U ovom slučaju ispisat će se "Not a number". Ako bi došlo do greške koja nije navedena u `rescue` bloku, tada bi se greška opet podigla.

Izbacivanje vrijednosti radi se pomoću naredbe `throw`, a hvata se pomoću `rescue`. Rade na sličnom principu kao i `throw/catch`.

impure.ex

```
defmodule Impure do
  def multiply_ten() do
    try do
      input = IO.gets("Give me a number in string: ")
      case Integer.parse(input) do
        :error -> throw {:error, "Not a number"}
        {number, _} -> Integer.to_string(number * 10)
      end
    catch
      {:error, hint} -> hint
    end
  end
end
```

Funkcija treba primiti cijeli broj kao string da bi isti broj umnožen 10 puta vratila opet kao string. Unutar `try` bloka su naredbe za koje se nadamo da će se izvršiti bez problema. Kad u `Integer.parse/2` dođe do greške jer nije unešen broj, baca se torka kojoj je pridijeljen atom `:error` i poruka. U `catch` bloku traži se upravo torka u kojoj se nalaze atom `:error`i

poruka te se u tom slučaju vraća poruka. Ako je u funkciji potreban samo jedan `try` blok, može ga se i izostaviti. U Elixiru se uglavnom pokušavaju izbjeći korištenja `catch`, `rescue`, `raise` i `throw`, ali to nije uvijek moguće.

Ako želimo kombinirati provjeru više klauzula, možemo koristiti komandu `with`.

impure.ex

```
defmodule Impure do
  def multiply() do
    with {number, _} <- factor("First"),
      {another_number, _} <- factor("Second")
    do
      Integer.to_string(number * another_number)
    else
      :error -> "Not numbers"
    end
  end

  defp factor(which) do
    IO.gets(which <> " factor to multiply: ")
    |> Integer.parse()
  end
end
```

Vratit će se rezultat iz do bloka ako sve klauzule pašu, u protivnom, ako ijedna ne odgovara, vratit će se vrijednost koja se dobila pridruživanjem i nije odgovarala klauzuli. Na ovaj način ne mora se provjeravati je li došlo do greške svaki put kad se poziva funkcija `factor` nego samo jednom. Ako vraćena vrijednost ne odgovara ni `with` ni `else` bloku, javit će se greška. Dobro je uvijek eksplicitno tražiti koja greška odgovara da bi se izbjegli bugovi koje je teško otkriti.

Poglavlje 2

Okruženje Phoenix

Phoenix [11] je okruženje za razvoj weba napisan u programskom jeziku Elixir. Koristi model-pogled-upravljač (eng. *model–view–controller*) ili skraćeno MVC obrazac s poslužiteljske strane. MVC je obrazac za dizajniranje softvera u kojemu se programska logika rastavlja na tri međusobno povezana dijela. Model je središnja komponenta koja direktno upravlja podacima i logikom aplikacije. Pogled je bilo kakva reprezentacija informacija. Upravljač prima ulaz i pretvara ga u naredbe modelu ili pogledu. Phoenix je razvijen za pružanje vrlo učinkovitih i skalabilnih web aplikacija. Uz zahtjev/odgovor funkcionalnost pruženu od Cowboy poslužitelja, nudi i komunikaciju u stvarnom vremenu pomoću svojstva kanala. Primjeri aplikacija koje koriste Phoenix nalaze se u [13] i [14].

2.1 Instalacija i upoznavanje

Phoenix treba Elixir. Za potrebe aplikacije koja će se graditi u ovom poglavlju poželjno je imati Elixir verzije 1.9.2 ili novije. Aplikacija će pružati korisnicima igranje `dame`, klasične društvene igre. Cjelokupna aplikacija dostupna je na [8]. Potreban će biti Hex, Elixirov upravitelj paketa, koji se može instalirati iz terminala komandom `mix local.hex`. Phoenix koristi Ecto za rad s bazama podataka, a Ecto koristi PostreSQL adapter pa je potrebno instalirati i PostreSQL praćenjem uputa na [1]. Poželjna je verzija 12 ili novija. Phoenix će koristiti webpack.js.org za kompajliranje JavaScript i CSS datoteka, a webpack koristi npm, Node.js-ov upravitelj paketa, da instalira svoje zavisnosti (eng. *dependencies*). Node.js se instalira praćenjem uputa na [10], a potrebna je verzija 12.14 ili novija. Phoenix ima svojstvo ponovnog učitanja uživo, (eng. *live reloading*), što znači da se promjenom datoteka automatski ponovno kompajliraju izmjenjene datoteke. Linux, za razliku od ostalih operativnih sustava, ne podržava Phoenixov live reloading, ali instalacijom alata inotify na [9] taj problem je rješen. Sam Phoenix instalira se iz terminala komandom `mix archive.install hex phx_new`.

Naredbom `mix phx.new checkers` izrađuje se novi projekt imena `Checkers`. Na pitanje `Fetch and install dependencies?[Yn]` odgovara se `y`. Potrebno je još samo unutar mape projekta postaviti bazu podataka za korištenje komandom `mix ecto.create` i pokrenuti server komandom `mix phx.server`. Korisničko ime i lozinka za PostgreSQL moraju odgovarati korisničkom imenu i lozinci u datoteci `config/dev.exs`. Novoj aplikaciji pristupa se Internet preglednikom na adresi http://localhost:4000/. Sadržaj u datoteci `lib/checkers_web/templates/page/index.html.eex` zamijenimo sljedećim kodom:

lib/checkers_web/templates/page/index.html.eex

```
<section class="phx-hero">
  <h1><%= gettext "Welcome to %{name}!", name: "Checkers
     " %></h1>
</section>
```

2.2 Upravljač i pogled

Prvi korak je izgradnja upravljača (eng. *controller*) koji će rukovati korisnicima. Kad preglednik uputi zahtjev upravljaču kroz krajnju točku (`lib/checkers_web/endpoint.ex`), dolazi u usmjernik (`lib/checkers_web/router.ex`) koji za dani URL pronalazi odgovarajući upravljač i njegovu akciju. Kontekst u Phoenixu je modul koji grupira funkcije zajedničke svrhe. Naš prvi kontekst bit će `Accounts` u kojem će se nalaziti funkcionalnosti za korisnike. Treba nam struktura koja predstavlja korisnika pa ćemo napraviti datoteku `lib/checkers/accounts/user.ex` sa sljedećim kodom:

lib/checkers/accounts/user.ex

```
defmodule Checkers.Accounts.User do
  defstruct [:id, :name, :username]
end
```

Definirana je struktura koja sadrži polja `id`, `name` i `username`, a može se napraviti s `%Checkers.Accounts.User{}`. Zatim u datoteci `lib/checkers/accounts.ex` definiramo kontekst `Accounts` sljedećim kodom:

lib/checkers/accounts.ex

```
defmodule Checkers.Accounts do
  @moduledoc """
  The Accounts context
  """
```

```
  alias Checkers.Accounts.User

  def list_users do
    [
      %User{id: "1", name: "Stjepan Hardcoded", username
         : "stjepanhardcoded"},
      %User{id: "2", name: "Hard Coded", username: "
         hardcoded"}
    ]
  end

  def get_user(id) do
    Enum.find(list_users(), fn map -> map.id == id end)
  end

  def get_user_by(params) do
    Enum.find(list_users(), fn map ->
      Enum.all?(params, fn {key, val} -> Map.get(map,
         key) == val end)
    end)
  end
end
```

Modul nam pruža par funkcija: `list_users` vraća listu svih korisnika, `get_user` vraća korisnika s odgovarajućim `id`-om, a `get_user_by` vraća korisnika s odgovarajućim atributima. Ako se želi pristupiti stranici koja prikazuje podatke korisnika, treba poslati zahtjev. Zahtjeve s određenog URL-a treba mapirati kodu koji udovoljava tom zahtjevu. To se radi u usmjerničkom sloju. Rute u Phoenixu su u datoteci `lib/checkers_web/router.ex`. Funkcija `scope` unutar `router.ex` određuje za koji URL će se zvati koja akcija. Dodajmo u funkciju `scope` `get` zahtjev za URL `user` i povezanu akciju `:show`:

lib/checkers_web/router.ex

```
scope "/", CheckersWeb do
  pipe_through :browser

  get "user/:id", UserController, :show
  get "/", PageController, :index
end
```

Ako usmjerniku pristigne `get` zahtjev s URL-a koji počinje na `/user/id` , gdje id označava neki broj, usmjernik će zvati `:show` funkciju modula `UserController` i proslijediti joj mapu s ključem/parom `"id"/id`. Upravljači su u `lib/checkers_web/controllers`. Napravimo datoteku `lib/checkers_web/controllers/user_controller.ex`:

lib/checkers_web/controllers/user_controller.ex

```
defmodule CheckersWeb.UserController do
  use CheckersWeb, :controller

  alias Checkers.Accounts
  alias Checkers.Accounts.User

  def show(conn, %{"id" => id}) do
    user = Accounts.get_user(id)
    render(conn, "show.html", user: user)
  end
end
```

`use CheckersWeb, :controller` označava da će se koristiti Phoenixov Controller API. `show` je ime akcije koju će usmjernik zvati u upravljaču, prosljeđujući sve potrebne informacije, u ovom slučaju mapu s ključem `"id"` i odgovarajućom vrijednošću. Dohvaćaju se podaci o korisniku i šalju funkciji `render` zajedno s imenom predloška. Da bi nam Phoenix imao što prikazati kad se pozove `show` potreban je odgovarajući `pogled` (eng. *view*). Pogled je modul koji sadrži funkcije koje pretvaraju podatke u format za korisnike. Te funkcije mogu biti definirane i iz predložaka. Predložak (eng. *template*) je funkcija kompajlirana iz datoteke koja sadrži čisti označni jezik (eng. *markup language*) poput HTML-a, i Elixir kod za procesiranje zamjeni i petlji. Pogledi su u `lib/checkers_web/views`. Napravimo datoteku `lib/checkers_web/views/user_view.ex`:

lib/checkers_web/views/user_view.ex

```
defmodule CheckersWeb.UserView do
  use CheckersWeb, :view
end
```

Kod ne radi puno, samo veže akciju `show` s odgovarajućim predloškom. Svi predlošci nalaze se u `lib/checkers_web/templates`. Jer je upravljač `user` zvao funkciju `render` predložak se traži u `lib/checkers_web/templates/user`.Ime predloška upravljač je također proslijedio pogledu, `show.html`. U tom slučaju, potrebno je napraviti datoteku `lib/checkers_web/templates/user/show_html.eex`:

lib/checkers_web/templates/user/show_html.eex

```
<h3><%= @user.username %></h3>
<h3><%= @user.name %></h3>
```

U predlošku su samo HTML oznake i Elixirov kod. Jezik predloška naziva se `EEx`, Embedded Elixir, i dio je Elixira. Predložak koristi vrijednost varijable `user` koja je određena u akciji `show` i prikazuje polja `username` i `name`. `<%= %>` označava kod koji će se zamijeniti odgovarajućim vrijednostima. Pomoću `@` pristupa se varijablama koje je upravljač proslijedio funkciji `render`. Ekstenzija `.eex` označava predložak koji će Phoenix kompajlirati u funkciju.

U ovom trenu može se pristupiti stranici http://localhost:4000/user/1 koja prikazuje `username` i `name` korisnika čiji je `id` 1. Kad se pozvala funkcija `render` u upravljaču, opcija `:layout` bila je postavljena na zadano, upravljač je prvo prikazao layout pogled iz datoteke `lib/hello_web/views/layout_view.ex` i njegov odgovarajući predložak `lib/hello_web/templates/layout/app.html.eex`. Layout pogled omogućuje konzistentne oznake na svim stranicama aplikacije bez dupliciranja koda.

2.3 Rad s bazom uz Ecto

Ecto [2] je omotač za relacijske baze podataka i omogućava developerima čitanje i očuvanje podataka. Sadrži svoj jezik upita kojim se mogu graditi slojeviti upiti. Ima i svojstvo zvano *changesetovi* koje sadrži sve promjene koje se žele obaviti na bazi. Stvaranjem projekta, Phoenix je generirao Ecto repozitorij, `lib/checkers/repo.ex`. Korisničko ime i lozinka u konfiguracijskoj datoteci `config/dev.exs` moraju odgovarati korisničkom imenu i lozinci za PostgreSQL. Repozitorij koristi PostgreSQL adapter. Ecto dopušta određivanje structa koji veže pojedina polja za polja u tablici baze. Zamijenimo kod u `lib/checkers/accounts/user.ex` sljedećim:

lib/checkers/accounts/user.ex

```
defmodule Checkers.Accounts.User do
  use Ecto.Schema
  import Ecto.Changeset

  schema "users" do
    field :name, :string
    field :username, :string

    timestamps()
  end
end
```

Modul koristi `use Ecto.Schema` i `import Ecto.Changeset` uvozi funkcije za kasniji rad s changesetovima. Makroi `schema` i field u isto vrijeme određuju tablicu baze i Elixirov struct. I tablica i struct imaju polja `:name` i `:username`, pa čak i `:id` koji se automatski definira kao primarni ključ. `timestamps()` izrađuje polja `inserted_at` i `updated_at`. Kako baza treba odgovarati strukturama u aplikaciji, treba se napraviti migracija koja mijenja bazu. Migracija se generira s `mix ecto.gen.migration create_users` gdje je `create_users` ime migracije. Migraciju treba definirati unutar novonastale datoteke `priv/repo/migrations/20200518074152_create_users.exs`:

priv/repo/migrations/20200518074152_create_users.exs

```
def change do
    create table(:users) do
      add :name, :string
      add :username, :string, null: false
      add :password_hash, :string

      timestamps()
    end

    create unique_index(:users, [:username])
  end
```

Nova tablica će se zvati `users`, imat će polja `name`, `username`, `password_hash` te automatski generirane `id`, `inserted_at` i `updated_at`. Tablica sadrži i jedinstveni index na `username`. Nakon definiranja migracije treba ju i pokrenuti s `mix ecto.migrate`. Kad je baza spremna, mogu se početi čuvati podaci u njoj. Komandom `iex -S mix` pokreće se ljuska preko koje možemo dodati novog korisnika

```
iex> alias Checkers.Repo
iex> alias Checkers.Accounts.User
Repo.insert(%User{name: "Stjepan", username: "stjepan"})
```

Sad kad postoje podaci u bazi, treba ih se i moći dohvatiti preko konteksta. U kontekstu `Accounts` zamijenimo sadržaj sljedećim kodom:

lib/checkers/accounts.ex

```
defmodule Checkers.Accounts do
  alias Checkers.Repo
  alias Checkers.Accounts.User

  def get_user(id) do
```

```
    Repo.get(User, id)
  end

  def get_user!(id) do
    Repo.get!(User, id)
  end

  def get_user_by(params) do
    Repo.get_by(User, params)
  end
end
```

Nove funkcije dohvaćaju korisnika preko `Ecto.Repo` funkcija. Nakon promjena u bazi, treba prekinuti i ponovno pokrenuti server. Pristupom stranici http://localhost:4000/user/1 mogu se vidjeti novi podaci. Aplikacija sada treba i mogućnost dodavanja novih korisnika. Potrebno je dodati novu funkciju u struct `User` koja će nam pomoći stvarati nove i mijenjati stare korisnike:

lib/checkers/accounts/user.ex

```
  def changeset(user, attrs) do
    user
    |> cast(attrs, [:name, :username])
    |> validate_required([:name, :username])
    |> validate_length(:username, min: 1, max: 32)
    |> unique_constraint(:username)
  end
```

Changesetovi enkapsuliraju cijeli proces primanja, promjene i validiranja podataka prije zapisa u bazu. Ova funkcija prima struct `User` i atribute. `cast` iz liste atributa označava `:name` i `:username` kao mogući korisnički unos. Funkcija `validate_required` određuje koji su atributi nužni. Funkcija `validate_length` određuje uvjete na duljinu određenog atributa. `unique_constraint` u ovom slučaju ne dozvoljava `:username` ako isti već postoji u bazi. Novu funkcionalnost koristit će kontekst `Accounts` novom funkcijom:

lib/checkers/accounts.ex

```
  def change_user(%User{} = user) do
    User.changeset(user, %{})
  end
```

Funkcija samo vraća novi changeset. U `UserController` treba dodati novu akciju koja će pozivati `change_user`:

lib/checkers_web/controllers/user_controller.ex

```
  def new(conn, _params) do
    changeset = Accounts.change_user(%User{})
    render(conn, "new.html", changeset: changeset)
  end
```

Funkcija će stvarati novi `User` struct i proslijediti ga predlošku, što znači da treba napraviti i novi odgovarajući predložak `lib/checkers_web/templates/user/new.html.eex`:

lib/checkers_web/templates/user/new.html.eex

```
<h1>Create new user</h1>
<%= form_for @changeset, Routes.user_path(@conn, :create
  ), fn f -> %>
    <%= if @changeset.action do %>
        <div class ="alert alert-danger">
        <p>Something went wrong! Check below</p>
        </div>
    <% end %>
    <div>
        <%= text_input f, :name, placeholder: "Name" %>
        <%= error_tag f, :name %>
    </div>
    <div>
        <%= text_input f, :username, placeholder: "
          Username" %>
        <%= error_tag f, :username %>
    </div>
    <%= submit "Create user" %>
<% end %>
```

Za stvaranje forme nismo koristili HTML oznake nego funkciju `form_for`, koja prima prima changeset, rutu i anonimnu funkciju. Anonimna funkcija prima podatke forme `f`. Gradi se forma s dva tekstualna unosa, za `name` i `username` korisnika i gumbom za podnošenje forme. `changeset` sadrži `action` koji upućuje na to koja akcija je bila pokušana na changesetu i na početku je prazan. Ako `action` nije prazan prikazat će se poruka i odgovarajuće greške pomoću `error_tag` funkcija. Funkcija `form_for` nalazi se u modulu `Phoenix.HTML`. Taj modul pruža rad s formama, generiranje linkova i sigurnost HTML-a. Modul je uvezen u datoteci `lib/checkers_web.ex` unutar funkcije `view`, koja definira što će se uvesti u pogled svaki put kad se iskoristi naredba `use`

`CheckersWeb, :view`. Trebat će nam i nova ruta za novu akciju upravljača. `scope` funkciju u `lib/checkers_web/router.ex` zamijenimo sljedećim kodom:

lib/checkers_web/router.ex

```
scope "/", CheckersWeb do
  pipe_through :browser

  resources "/user", UserController, only: [:show, :new,
    :create]
  get "/", PageController, :index
end
```

Koristi se funkcija `resources` koja dodaje akcije za CRUD operacije (eng. *Create, Read, Update, Delete*), s tim da preciziramo koje akcije želimo. Popis dostupnih ruta, može se dobiti pokretanjem `mix phx.routes`. Sada postoje ruta, upravljač i predložak za stvaranje novih korisnika pa treba napisati i funkciju koja će zapravo stvarati korisnike. U `lib/checkers/accounts.ex` treba dodati funkciju:

lib/checkers/accounts.ex

```
  def create_user(attrs \\ %{}) do
    %User{}
    |> User.changeset(attrs)
    |> Repo.insert()
  end
```

Funkcija počinje od praznog structa `User`, koristi na njemu funkciju `User.changeset` i ubacuje korisnika u repozitorij. Preostaje samo pozvati funkciju novom akcijom iz upravljača `lib/checkers_web/controllers/user_controller.ex`:

lib/checkers_web/controllers/user_controller.ex

```
def create(conn, %{"user" => user_params}) do
  case Accounts.create_user(user_params) do
    {:ok, user} ->
      conn
        |> put_flash(:info, "#{user.name} created!")
        |> redirect(to: Routes.page_path(conn, :index))
    {:error, %Ecto.Changeset{} = changeset} ->
      render(conn, "new.html", changeset: changeset)
  end
end
```

Akcija zove funkciju `Accounts.create_user` i ako uspije napraviti korisnika dodaje poruku koja će se ispisati i preusmjerava na početnu stranicu, a u protivnom prikaže se ista stranica i razlog zbog čega je došlo do greške. U ovom trenu može se posjetiti http://localhost:4000/user/new i stvoriti novog korisnika.

2.4 Autentifikacija i prijava

Vrijeme je za uvesti autentifikaciju korisnika. Koristit će se `Pbkdf2` heširanje lozinki pa je potrebno dodati `:pbkdf2_elixir` u zavisnosti aplikacije u `mix.exs` datoteci na sljedeći način:

mix.exs

```
  defp deps do
    [
        ...,
      {:pbkdf2_elixir, "~> 1.0"}
    ]
  end
```

Nove zavisnosti dohvaćaju se naredbom `mix.deps.get`. U shemi korisnika dodamo dva nova polja:

lib/checkers/accounts/user.ex

```
    field :password, :string, virtual: true
    field :password_hash, :string
```

i dvije nove funkcije:

lib/checkers/accounts/user.ex

```
def registration_changeset(user, params) do
  user
  |> changeset(params)
  |> cast(params, [:password])
  |> validate_required([:password])
  |> validate_length(:password, min: 6, max: 64)
  |> put_pass_hash()
end

defp put_pass_hash(changeset) do
  case changeset do
```

```
      %Ecto.Changeset{valid?: true, changes: %{password:
         pass}} ->
        put_change(changeset, :password_hash, Pbkdf2.
           hash_pwd_salt(pass))

      _ ->
        changeset
    end
  end
```

Dodana su dva nova polja, `password` i `password_hash`. `virtual` označava da se polje pojavljuje samo u structu, ne i u bazi. U modulu su definirane i dvije nove funkcije. `registration_changeset` stvara novi changeset, za kojega korisnik može unijeti `password`, koji je obavezan i mora imati određenu duljinu te pomoću `put_pass_hash` valjane lozinke hešira i sprema u changeset, a u slučaju nevaljane lozinke vraća cijeli changeset. Stari korisnici nemaju postavljene lozinke pa ćemo im ih dodati preko ljuske koju palimo komandom `iex -S mix`:

```
iex> alias Checkers.Repo
iex> alias Checkers.Accounts.User
iex> for u <- Repo.all(User) do Repo.update!(
User.registration_changeset(u, %{password: "password"})) end
```

Treba omogućiti i novim korisnicima da si sami izaberu lozinku. Dodajmo nove funkcije u kontekst `lib/checkers/accounts.ex`:

lib/checkers/accounts.ex

```
  def change_registration(%User{} = user, params) do
    User.registration_changeset(user, params)
  end

  def register_user(attrs \\ %{}) do
    %User{}
    |> User.registration_changeset(attrs)
    |> Repo.insert()
  end
```

`change_registration` samo vraća novi `registration_changeset`. `register_user` poziva `User.registration_changeset` i dobiveni changeset sprema u repozitorij, čime obavlja registraciju korisnika. U upravljaču `UserContoller` potrebno je izmjeniti dvije linije da bi koristili nove funkcionalnosti:

lib/checkers_web/controllers/user_controller.ex

```
changeset = Accounts.change_user(%User{})
```

treba zamijeniti s:

lib/checkers_web/controllers/user_controller.ex

```
changeset = Accounts.change_registration(%User{}, %{})
```

I liniju:

lib/checkers_web/controllers/user_controller.ex

```
case Accounts.create_user(user_params) do
```

treba zamijeniti s:

lib/checkers_web/controllers/user_controller.ex

```
case Accounts.register_user(user_params) do
```

U predložak još samo treba dodati unos lozinke prije gumba za podnošenje forme:

lib/checkers_web/templates/user/new.html.eex

```
    <div>
        <%= password_input f, :password, placeholder: "
           Password" %>
        <%= error_tag f, :password %>
    </div>
```

U ovom trenutku ako se posjeti http://localhost:4000/user/new može se stvoriti novog korisnika s lozinkom. Vrijeme je za implementirati autentifikaciju kako bi se korisnici mogli prijavljivati i odjavljivati. Autentifikacija će se obavljati pomoću *pluga*. Svaki plug prima `conn`, transformira ga i šalje dalje. Kada želimo da se plug pojavljuje u više modula koristi se module plug, koji mora implementirati dvije funkcije, `init` i `call`. U novu datoteku modula `Auth` unesemo sljedeći kod:

lib/checkers_web/controllers/auth.ex

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

  def init(opts), do: opts

  def call(conn, _opts) do
    user_id = get_session(conn, :user_id)
```

```
    user = user_id && Checkers.Accounts.get_user(user_id
  )
    assign(conn, :current_user, user)
  end

  def login(conn, user) do
    conn
    |> assign(:current_user, user)
    |> put_session(:user_id, user.id)
    |> configure_session(renew: true)
  end

  def logout(conn) do
    configure_session(conn, drop: true)
  end
end
```

Funkcija `init` mora postojati da dopusti opcije u vremenu kompajliranja. `call` traži u sesiji `user_id` i zove funkciju `assign` koja transformira `conn` spremajući `user_id` (ili `nil` ako nije pronađen `user_id` u sesiji) u `conn`. `login` će u `conn` dodati `user_id`, staviti ga i u sesiju, konfigurirati ju i, zbog `renew: true`, poslati novi kolačić (eng. *cookie*) klijentu radi sigurnosnih razloga. `logout` ispušta cijelu sesiju. Plug treba dodati u usmjernik na sljedeći način:

lib/checkers_web/router.ex

```
  pipeline :browser do
    ...
    plug CheckersWeb.Auth
  end
```

Kako postoji informacija je li u `conn` spremljen `user_id`, može se ograničiti pristup nekim stranicama. Dodajmo sljedeću plug funkciju u upravljač `UserController`:

lib/checkers_web/controllers/user_controller.ex

```
defp authenticate(conn, _opts) do
  if conn.assigns.current_user do
    conn
  else
    conn
    |> put_flash(:error, "You must be logged in to
       access that page")
```

```
    |> redirect(to: Routes.page_path(conn, :index))
    |> halt()
  end
end
```

Da bi funkcija bila plug, mora primati `conn` i opcije, a vraćati `conn`. U slučaju da ne postoji `user_id` u sesiji, funkcija će pokazati poruku greške i preusmjeriti na početnu stranicu te zaustaviti daljnju transformaciju `conn`. Za korištenje novog pluga, treba upravljaču reći da ga koristi. U `UserController` nakon poziva `alias` treba ubaciti liniju:

lib/checkers_web/controllers/user_controller.ex

```
  plug :authenticate when action in [:show]
```

Za prijavu korisnika nakon registracije u upravljaču `UserController` dodajmo još na pravo mjesto liniju `|> CheckersWeb.Auth.login(user)`:

lib/checkers_web/controllers/user_controller.ex

```
case Accounts.register_user(user_params) do
  {:ok, user} ->
    conn
    |> CheckersWeb.Auth.login(user)
    |> put_flash(:info, "#{user.name} created!")
```

Kontekst `Accounts` je savršeno mjesto za definiranje funkcije koja autenticira dano korisničko ime i lozinku:

lib/checkers/accounts.ex

```
def authenticate_by_username_and_password(username,
   given_pass) do
  user = get_user_by(username: username)
  cond do
    user && Pbkdf2.verify_pass(given_pass, user.
       password_hash) ->
      {:ok, user}
    user ->
      {:error, :unauthorized}
    true ->
      Pbkdf2.no_user_verify()
      {:error, :not_found}
  end
end
```

Prvo se traži korisnika zadanog korisničkog imena. Ako je korisničko ime krivo, simulira se provjera lozinke u slučaju napada na aplikaciju i vraća torka s greškom. Ako lozinka odgovara korisničkom imenu vraća se torka s podacima korisnika, a u suprotnom vraća se torka s greškom. Potrebno je još napraviti novu stranicu za prijavu korisnika. Dodajmo nove rute u usmjernik:

lib/checkers_web/router.ex

```
    resources "/sessions", SessionController, only: [:
      new, :create, :delete]
```

Dodaju se tri REST rute, `GET /sessions/new` prikazuje formu za prijavu, `POST /sessions` za prijavu i `DELETE /sessions/:id` za odjavu. Potreban je upravljač `SessionController` koji će imati navedene akcije:

lib/checkers_web/router.ex

```
defmodule CheckersWeb.SessionController do
  use CheckersWeb, :controller

  def new(conn, _) do
    render(conn, "new.html")
  end

  def create(conn, %{"session" => %{"username" =>
     username, "password" => password}}) do
    case Checkers.Accounts.
       authenticate_by_username_and_password(username,
       password) do
      {:ok, user} ->
        conn
        |> CheckersWeb.Auth.login(user)
        |> put_flash(:info, "You have logged in!")
        |> redirect(to: Routes.page_path(conn, :index))

      {:error, _reason} ->
        conn
        |> put_flash(:error, "Invalid login input")
        |> render("new.html")
    end
  end
```

```
  def delete(conn, _) do
    conn
    |> CheckersWeb.Auth.logout()
    |> redirect(to: Routes.page_path(conn, :index))
  end
end
```

Akcija `new` prikazivat će formu za prijavu. `create` će preuzeti iz forme uneseno korisničko ime i lozinku i provjeriti ih, te će ako je provjera uspješna, prijaviti korisnika, prikazati poruku i preusmjeriti ga na početnu stranicu, a u slučaju neuspjeha prikazati na istoj stranici poruku greške. `delete` će pozvati funkciju za odjavljivanje i preusmjeriti na početnu stranicu. Potrebno je napraviti i pogled `SessionView` za upravljač:

lib/checkers_web/views/session_view.ex

```
defmodule CheckersWeb.SessionView do
  use CheckersWeb, :view
end
```

Svaki pogled treba i svoj predložak:

lib/checkers_web/templates/session/new.html.eex

```
<h1>Login</h1>
<%= form_for @conn, Routes.session_path(@conn, :create),
    [as: :session], fn f -> %>
    <div>
        <%= text_input f, :username, placeholder: "
           Username" %>
    </div>
    <div>
        <%= password_input f, :password, placeholder: "
           Password" %>
    </div>
    <%= submit "Log in" %>
<% end %>
```

Predložak se sastoji od forme koja prima dva unosa, korisničko ime i lozinku, i gumb za podnošenje forme. Forma umjesto changeseta šalje `conn`, jer u ovom slučaju forma nije poduprta changesetom. Sve što je preostalo je ponuditi korisnicima prijavu i odjavu preko poveznica. U layout predložak u <nav> sekciju potrebno je dodati sljedeći kod:

lib/checkers_web/templates/layout/app.html.eex

```
<nav role="navigation">
  <ul>
    <%= if @current_user do %>
    <li><%= link @current_user.username, to: Routes.
       user_path(@conn, :show, @current_user.id) %></li>
    <li><%= link "Log out", to: Routes.session_path(
       @conn, :delete, @current_user), method: "delete"
       %></li>
    <% else %>
    <li><%= link "Register", to: Routes.user_path(@conn,
        :new) %></li>
    <li><%= link "Log in", to: Routes.session_path(@conn
       , :new) %></li>
    <% end %>
  </ul>
</nav>
```

U slučaju prijavljenog korisnika prikazuju se poveznice na njegovu stranicu i za odjavu. U protivnom, prikazuju se poveznice za registraciju i prijavu. Pri izradi poveznice koristi se funkcija `link` iz modula `Phoenix.HTML`. Posjetom http://localhost:4000 mogu se vidjeti nove izmjene.

2.5 Kanali i komunikacija

Treba nam stranica na kojoj će se korisnici moći okupiti i pozivati jedan drugoga u igru. Dodajmo rutu za nju u usmjernik:

lib/checkers_web/router.ex

```
    get "/lobby", LobbyController, :show
```

Napravimo novi upravljač `LobbyController`:

lib/checkers_web/controllers/lobby_controller.ex

```
defmodule CheckersWeb.LobbyController do
  use CheckersWeb, :controller

  def show(conn, _) do
    user_id = get_session(conn, :user_id)
    render(conn, "show.html", user_id: user_id)
  end
```

```
  end
```

Upravljač samo dohvaća iz sesije `user_id` i šalje ga pogledu. Novom upravljaču treba i novi pogled `LobbyView`:

lib/checkers_web/views/lobby_view.ex

```
defmodule CheckersWeb.LobbyView do
  use CheckersWeb, :view
end
```

Novom pogledu treba i novi predložak:

lib/checkers_web/templates/lobby/show.html.eex

```
<h2>Welcome to checkers lobby.</h2>
<div class="row">
    <div class="col">
        <%# <div id="lobby-chat">
        </div> %>
        <%= content_tag :div, id: "lobby-chat", data: [
           user_id: @user_id] do %>
        <% end %>
        <div id="message-controls">
            <input type="text" id="message-input"
               placeholder="Send message">
            <%# textarea %>
            <button id="send-message" type="submit">Send
               </button>
        </div>
    </div>
</div>
```

U predložak smo stavili uglavnom HTML oznake među kojima je na kojem će biti prikazane buduće poruke, <input> u kojega će se upisivati poruke za slanje i gumb za slanje. Koristimo i dodatnu funkciju `content_tag` koja će napraviti element s `data-user-id` atributom. Dodajmo sljedeći kod na početnu stranicu:

lib/checkers_web/templates/page/index.html.eex

```
  <%= if @current_user do %>
    <%= link "PLAY", to: Routes.lobby_path(@conn, :show)
       %>
  <% else %>
```

```
    <h2>Register and login to play!</h2>
  <% end %>
```

Ako je korisnik prijavljen, moći će stisnuti na poveznicu za lobby. U protivnom će pisati naslov da se treba registrirati kako bi igrao. Dodajmo i poveznicu za lobby ako je korisnik prijavljen, u <nav> sekciju layout predloška:

lib/checkers_web/templates/layout/app.html.eex

```
    %= if @current_user do %
      <li><%= link "Lobby", to: Routes.lobby_path(@conn,
          :show) %></li>
```

U upravljaču `SessionController` , u funkciji `create` liniju:

lib/checkers_web/controllers/session_controller.ex

```
|> |> redirect(to: Routes.page_path(conn, :index))
```

zamijenimo s:

lib/checkers_web/controllers/session_controller.ex

```
|> redirect(to: Routes.lobby_path(conn, :show))
```

Korisnika će nakon registracije usmjeravati na lobby. Uobičajeno je da pretražitelj šalje zahtjev poslužitelju koji zatim vraća odgovor. U Phoenixu klijent može direktno kontaktirati poslužitelja preko kanala. Kanal šalje i prima poruke te čuva stanje. Poruke su događaji (eng. `events`), a stanje se čuva u structu `socket`. U svakom kanalu obavlja se razgovor o jednoj temi, kao što je soba za razgovor ili instanca igre. Kako bi se spojio na kanal, klijent mora uspostaviti vezu s utičnicom (eng. `socket`). Phoenix je već napravio datoteku `assets/js/socket.js`. Zamijenimo sadržaj te datoteke sljedećim:

assets/js/socket.js

```
import {Socket} from "phoenix"
let socket = new Socket("/socket", {params: {token:
   window.userToken}})
socket.connect()
export default socket
```

Uvozi se objekt `Socket` , instancira nova utičnica i omogućuje spajanje. Tema kanala je zapravo identifikator po kojem se prepoznaje kanal. U već postojeći `UserSocket` dodajmo na početak datoteke identifikator za kanal koji ćemo napraviti:

lib/checkers_web/channels/user_socket.ex

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

  ## Channels
  channel "lobby", CheckersWeb.LobbyChannel
```

`UserSocket` već sadrži funkcije `connect` i `id`. `connect` odlučuje hoće li se uspostaviti veza. `id` daje mogućnost identificiranja utičnice na temelju stanja spremljenog u samoj utičnici. Novi kanal će imati identifikator `lobby`, a zvat će se `LobbyChannel`:

lib/checkers_web/channels/lobby_channel.ex

```
defmodule CheckersWeb.LobbyChannel do
  use CheckersWeb, :channel

  def join("lobby", _params, socket) do
    {:ok, socket}
  end

  alias Checkers.Accounts

  def handle_in(event, params, socket) do
    user = Accounts.get_user!(params["userId"])
    handle_in(event, params, user, socket)
  end

  def handle_in("new_message", params, user, socket) do
    broadcast!(socket, "new_message", %{
      user: %{username: user.username},
      body: params["body"]
    })

    {:reply, :ok, socket}
  end
end
```

`join` samo omogućuje spajanje na kanal. `handle_in` rukuje dolaznim porukama. Dohvatit će korisnika koji je poslao poruku i proslijediti ga zajedno s ostalim pristiglim podacima. `broadcast` će zatim poslati događaj svim korisnicima u kanalu. Još treba napraviti dio koji će slati poruke. Phoenix koristi `webpack`, alat za izgradnju, transformaciju i minimiziranje JavaScript i CSS koda. U direktoriju `asset` stavljaju se CSS i JavaScript datoteke u

svoje direktorije. Kad se pokrene aplikacija u Phoenixu, webpack će automatski pokupiti sadržaj JavaScript datoteka i skupiti ih u datoteku `priv/static/js/app.js`, koja se učitava u `app` predlošku. Napravimo objekt `Lobby` koji će biti zadužen za slanje i prikaz poruka:

assets/js/lobby.js

```
let Lobby = {
    init(socket, element) {if(!element) {return}
        socket.connect()
        let userId = element.getAttribute("data-user-id"
           )
        this.onReady(userId, socket)
    },

    onReady(userId, socket){
        let chat = document.getElementById("lobby-chat")
        let msgInput = document.getElementById("message-
           input")
        let sendButton = document.getElementById("send-
           message")
        let channel_lobby = socket.channel("lobby")

        msgInput.addEventListener("keypress" , e => {
            if (e.keyCode == 13 && msgInput.value.length
                > 0){
                this.SendMessage(channel_lobby, userId)
            }
        })
        sendButton.addEventListener("click", e => {
            if (msgInput.value.length > 0)
                this.SendMessage(channel_lobby, userId)
        })
        channel_lobby.on("new_message", resp => {
            this.RenderMessage(chat, resp)
        })
        channel_lobby.join().receive("ok", resp =>
           console.log("joined lobby", resp))
            .receive("error", reason => console.log("
               failed lobby", reason))
    },
```

```
    SendMessage(channel_lobby, userId){
        let msgInput = document.getElementById("message-
            input")
        let payload = {body: msgInput.value, userId:
            userId}
        channel_lobby.push("new_message", payload)
                .receive("error", e => console.log(e))
        msgInput.value = ""
    },

    esc(str){
        let div = document.createElement("div")
        div.appendChild(document.createTextNode(str))
        return div.innerHTML
    },

    RenderMessage(chat, {user, body}){
        let template = document.createElement("div")
        template.innerHTML =`
        <b>${this.esc(user.username)}</b>: ${this.esc(
            body)}
        `
        chat.appendChild(template)
        chat.scrollTop = chat.scrollHeight
    }
}
export default Lobby
```

Definirana je funkcija `init` koja izvlači `user_id` iz atributa i započinje vezu s utičnicom. `onReady` će se pobrinuti za potrebne varijable, slušatelje događaja i za spajajanje na kanal. Neprazne poruke slat će se kanalu stiskom gumba enter ili `Send`. `RenderMessage` prikazivat će poruke dobivene iz kanala zajedno s imenom korisnika koji ih je poslao. `esc` se brine da primljene poruke budu sigurne za ubacivanje na stranicu. Iz datoteke `assets/js/app.js` pokreće se sav JavaScript kod pa u nju treba uvesti objekt `Lobby` i utičnicu:

assets/js/app.js

```
import socket from "./socket"
import Lobby from "./lobby"
```

```
Lobby.init(socket, document.getElementById("lobby-chat")
  )
```

Dodajmo još i CSS datoteku `assets/css/lobby.css` koja je dostupna na [8] te ju uvezimo u glavnu CSS datoteku:

assets/css/app.css

```
@import "./lobby.css";
```

U ovom trenu, kada se korisnik prijavi, preusmjerit će ga na lobby te će se tamo moći dopisivati.

Autentifikacija sesije ima smisla za aplikacije koje koriste zahtjev/odgovor. Za kanale, autentifikacija žetona (eng. *token*) radi bolje jer je veza dugotrajna. Token autentifikacija je mehanizam koji svakom korisniku dodijeljuje jedinstveni žeton. Phoenix u tu svrhu pruža svoj `Phoenix.Token`, kojeg treba generirati za korisnika i proslijediti utičnici. Žeton se može izložiti klijentskoj strani u layoutu na sljedeći način:

assets/css/app.css

```
    </main>
    <script>window.userToken = "<%= assigns[:user_token]
        %>"</script>
    <script type="text/javascript" src="<%= Routes.
       static_path(@conn, "/js/app.js") %>"></script>
  </body>
```

Dosada je autentifikaciju koristio samo upravljač `UserController`, a treba biti dostupna svuda pa treba napraviti par izmjena. Iz upravljača `UserController` prebacit ćemo funkciju `authenticate` u modul `Auth`, nazvati ju `authenticate_user` i definirati ju javnom, a ne privatnom. Iz `Phoenix.Controller` uvest ćemo funkcije `put_flash` i `redirect` te koristiti pomoćne funkcije za rute:

lib/checkers_web/controllers/auth.ex

```
  import Phoenix.Controller
  alias CheckersWeb.Router.Helpers, as: Routes

  def authenticate_user(conn, _opts) do
    if conn.assigns.current_user do
      conn
    else
      conn
```

```
      |> put_flash(:error, "You must be logged in to
         access that page")
      |> redirect(to: Routes.page_path(conn, :index))
      |> halt()
    end
  end

  defp put_current_user(conn, user) do
    token = Phoenix.Token.sign(conn, "user socket", user
       .id)

    conn
    |> assign(:current_user, user)
    |> assign(:user_token, token)
  end
```

Nova funkcija `put_current_user` dodaje još i žeton u `conn`. Također u `Auth`, funkciji `call` linije:

lib/checkers_web/controllers/auth.ex

```
user = user_id && Checkers.Accounts.get_user(user_id)
assign(conn, :current_user, user)
```

zamijenimo s:

lib/checkers_web/controllers/auth.ex

```
cond do
  user = conn.assigns[:current_user] ->
    put_current_user(conn, user)

  user = user_id && Checkers.Accounts.get_user(user_id)
     ->
    put_current_user(conn, user)

  true ->
    assign(conn, :current_user, nil)
end
```

I u funkciji `login` liniju:

lib/checkers_web/controllers/auth.ex

```
|> assign(:current_user, user)
```

zamijenimo s:

lib/checkers_web/controllers/auth.ex

```
|> put_current_user(user)
```

Kako će se sad autentifikacija moći provesti bilo gdje u aplikaciji, modul treba podijeliti svim upravljačima i usmjernicima. Uvest ćemo ga na dva mjesta u modul `CheckersWeb`, u funkcije `controller` i router :

lib/checkers_web.ex

```
import CheckersWeb.Auth, only: [authenticate_user: 2]
```

U upravljaču `UserController` potrebno je još zamijeniti liniju:

lib/checkers_web/controllers/user_controller.ex

```
plug :authenticate when action in [:show]
```

sa sljedećom:

lib/checkers_web/controllers/user_controller.ex

```
plug :authenticate_user when action in [:show]
```

Želimo ograničiti pristup i lobbyju pa u `LobbyController` dodajemo:

lib/checkers_web/controllers/lobby_controller.ex

```
plug :authenticate_user when action in [:show]
```

U `assets/js/socket.js` Phoenix je već sam pripremio da će utičnica u konstruktoru primati žeton pa je preostalo samo verificirati žeton na spajanju. Zamijenimo sve osim kanala u modulu `UserSocket` sljedećim kodom:

lib/checkers_web/channels/user_socket.ex

```
@max_age 60*60*24

def connect(%{"token" => token}, socket, _connect_info)
   do
  case Phoenix.Token.verify(socket, "user socket", token
     , max_age: @max_age) do
    {:ok, user_id} ->
      {:ok, assign(socket, :user_id, user_id)}
```

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

  def connect(_params, _socket, _connect_info), do: :error
  def id(socket), do: "user_socket:#{socket.assigns.
     user_id}"
```

Definirana je duljina trajanja žetona. Verificira se žeton i ako je valjan, sprema se `user_id` u utičnicu te se uspostavlja veza, a u suprotnom se odbija pokušaj veze. Kako bi koristili `user_id` iz utičnice, u kanalu `LobbyChannel` liniju:

lib/checkers_web/channels/lobby_channel.ex

```
user = Accounts.get_user!(params["userId"])
```

zamijenimo sljedećom linijom:

lib/checkers_web/channels/lobby_channel.ex

```
user = Accounts.get_user!(socket.assigns.user_id)
```

Praćenje koji je korisnik na stranici možda se čini lagano, no kad je aplikacija distribuirana na više poslužitelja postaje ozbiljan problem. `Phoenix.Presence` pruža praćenje prisutnosti u procesima i kanalima. Snaga modula `Presence` je u tome kako se lagano postavlja i koristi. Kako je baziran na Elixiru, ne moraju se dodavati nove zavisnosti. Za generirati modul koristi se naredba `mix phx.gen.presence`. Nakon generiranja modula treba ga dodati u stablo nadzora (eng. `supervision tree`) u `lib/checkers/application.ex` na sljedeći način:

lib/checkers/application.ex

```
children = [
  ...,
  CheckersWeb.Presence
]
```

Sad kad je `Presence` postavljen za korištenje, krenimo pratiti koji su korisnici trenutno u kanalu lobby. U `LobbyChannel` dodajmo sljedeću funkciju:

lib/checkers_web/channels/lobby_channel.ex

```
def handle_info(:after_join, socket) do
```

```
    push(socket, "presence_state", CheckersWeb.Presence.
       list(socket))
    {:ok, _} = CheckersWeb.Presence.track(socket, socket.
       assigns.user_id, %{device: "browser"})
    {:noreply, socket}
  end
```

U funkciju `join` na početku dodajmo liniju:

lib/checkers_web/channels/lobby_channel.ex

```
    send(self(), :after_join)
```

Spajanjem na kanal, pošalje se samom sebi poruka `:after_join` koju procesira funkcija `handle_info`. `handle_info` zove `Presence.Track` koja obavlja velik dio posla. Funkcija prima utičnicu, ključ koji će pratiti i mapu metapodataka. Ključ je jedinstveni `user_id`. U mapu stavljamo da je korisnik pristupio stranici s preglednika. `Presence` će čuvati ove podatke dok god je korisnik na vezi. Kad se prati prisutnost, zapravo se traži od Phoenixa da prati poruke o korisnikovim dolascima i odlascima. Dodajmo u lobby predložak HTML elemente u kojima ćemo pratiti korisnike na vezi:

lib/checkers_web/templates/lobby/show.html.eex

```
<div class="col">
  <h3>Online players</h3>
  <div id="online-players">
    <ul id="online">
      </ul>
  </div>
</div>
```

Prazna lista će se puniti JavaScript kodom. U JavaScript kod treba uvesti `Presence` u prvoj liniji datoteke `assets/js/lobby.js`:

assets/js/lobby.js

```
import {Presence} from "phoenix"
```

U istoj datoteci na kraj funkcije `onReady` dodajemo kod koji će koristiti `Presence`:

assets/js/lobby.js

```
let userList = document.getElementById("online")
let presence = new Presence(channel_lobby)
presence.onSync(() => {
```

```
    userList.innerHTML = presence.list((id, {user: user})
       => {
      if (userId == id) return
      return `<button id="invite-${id}">${user.username}</
         button>`}).join(" ")
})
```

Dodani kod instancira objekt `Presence` i funkcijom `onSync` na korisnikov dolazak ili odlazak, generira listu korisnika kao listu gumbova na koje će se kasnije stiskati da se igrača pozove u igru. Korisniku se neće prikazati gumb za samog sebe. Koristi se funkcija `presence.list` koja grupira korisnikove višestruke prisutnosti u jedan objekt, kako se korisnik ne bi prikazao više puta ako je prijavljen na različitim preglednicima. `presence.list` prima mapu koja zasad još nema ključ `user` u sebi, ali taj dio slijedi. Prvo će trebati funkcija u kontekstu `Accounts` koja dohvaća korisnike za zadanu listu id-eva:

lib/checkers/accounts.ex

```
import Ecto.Query
def list_users_with_ids(ids) do
  Repo.all(from(u in User, where: u.id in ^ids))
end
```

Koristi se Ecto upit da se dohvate korisnici čiji je id u zadanoj listi. Zatim u modul `Presence` treba dodati sljedeću funkciju:

lib/checkers_web/channels/presence.ex

```
def fetch(_topic, entries) do
  users =
    entries
    |> Map.keys()
    |> Checkers.Accounts.list_users_with_ids()
    |> Enum.into(%{}, fn user ->
      {to_string(user.id), %{username: user.username}}
    end)
  for {key, %{metas: metas}} <- entries, into: %{} do
    {key, %{metas: metas, user: users[key]}}
  end
end
```

Funkcija `fetch` prima mapu parova `user_id/session_metadata` i prosljeđuje ključeve `user_id` funkciji koja će vratiti korisnike te će se u mapu za svaki `user_id` spremiti odgovarajući `username`. Zatim se originalnu mapu metapodataka nadopuni korisničkim

imenima. Ako se u ovom trenu posjeti http://localhost:4000/lobby s dva različita prijavljena korisnika, svakome od njih u listi prijavljenih bit će prikazan ovaj drugi.

Sad kad korisnici mogu vidjeti da su drugi korisnici na vezi, želimo im omogućiti da ih pozivaju u igru. Dodajmo novi kanal preko kojega će se obavljati ta komunikacija u `UserSocket`:

lib/checkers_web/channels/user_socket.ex

```
  channel "user:*", CheckersWeb.UserChannel
```

Identifikator za kanal će biti `user` popraćen s id-om korisnika na čiji kanal se spaja. Treba napraviti i novu datoteku kanala `UserChannel`:

lib/checkers_web/channels/user_channel.ex

```
defmodule CheckersWeb.UserChannel do
  use CheckersWeb, :channel

  def join("user:" <> user_id, _params, socket) do
    {:ok, assign(socket, :user_id, String.to_integer(
       user_id))}
  end

  alias Checkers.Accounts

  def handle_in("invite", params, socket) do
    user = Accounts.get_user!(params["sourceId"])
    broadcast!(socket, "invite", %{
      user: %{username: user.username},
      source_id: params["sourceId"],
      dest_id: params["destId"]
    })
    {:reply, :ok, socket}
  end

  def handle_in("reject", params, socket) do
    broadcast!(socket, "reject", %{
      source_id: params["sourceId"],
      dest_id: params["destId"]
    })
    {:reply, :ok, socket}
  end
```

```
  def handle_in("cancel", params, socket) do
    broadcast!(socket, "cancel", %{
      source_id: params["sourceId"],
      dest_id: params["destId"]
    })
    {:reply, :ok, socket}
  end
end
```

Spajanjem na kanal, spremit će se u utičnicu `user_id`. Primanjem događaja `invite` poslat će se poruka `invite` s korisničkim imenom pozivatelja i id-evima pozivatelja i pozvanika. Primanjem događaja `reject` ili `cancel` šalju se id-evi pozivatelja i pozvanika s porukom `reject` ili `cancel`. Korisnik mora moći prihvatiti ili odbiti poziv pa dodajmo na početak datoteke predloška lobby sljedeće:

lib/checkers_web/templates/lobby/show.html.eex

```
<div id="waitingDialog" class="modal">
  <div class="modal-content">
    <p> Waiting for an answer</p>
      <button id="cancelBtn">Cancel</button>
  </div>
</div>
<div id="inviteDialog" class="modal">
  <div class="modal-content">
    <p id="invite-text"></p>
    <button id="acceptBtn">Yes</button>
    <button id="rejectBtn">No</button>
  </div>
</div>
```

Dodan je prozor za čekanje koji ima gumb za prekid koji će se moći stisnuti ako korisnik želi ipak odustati od pozivanja u igru. Dodan je i prozor za odgovor na poziv u igru s gumbima za prihvatiti i odbiti. Preostaje nam samo, u JavaScript kodu, pobrinuti se da sve radi. U datoteku `lobby.js` prije funkcije `presence.onSync` potrebno je dodati:

assets/js/lobby.js

```
let channel_invite = null
let available = true
let channel_user = socket.channel("user:" + userId)
let canceled = false
```

```
let timeout = null
let inviterId = null
```

Dodane su varijable koje će pomoći u rukovanju pozivima. Također na kraj funkcije presence.onSync potrebno je dodati:

assets/js/lobby.js

```
presence.list((id) => {
  if (userId == id) return
  let button = document.getElementById(`invite-${id}`)
  button.addEventListener("click", e => {
  available = false
    channel_invite = this.Invite(userId, id, socket)
  })
})
```

Klikom na gumb koji sadrži ime igrača, navedeni igrač pozvat će se u igru. Zatim na kraj funkcije onReady potrebno je dodati:

assets/js/lobby.js

```
channel_user.on("invite", resp => {
  if (resp.dest_id != userId) return
  if (inviterId != null || !available) {
      console.log(`invite reject ${resp.user.username}
         busy`)
      let payload = {sourceId: userId, destId: resp.
         source_id}
      channel_user.push("reject", payload)
          .receive("error", e => console.log(e))
      return
  }
  available = false
  inviterId = resp.source_id
  let inviteDialog = document.getElementById("
     inviteDialog")
  let inviteText = document.getElementById("invite-text"
     )
  inviteText.innerHTML = `${resp.user.username} invited
     you to a game.
      <br>
      Do you wish to accept?`
```

```
    inviteDialog.style.display = "block";

    timeout = setTimeout(function() {
        console.log("reject because of timeout")
        document.getElementById("rejectBtn").click()
    }, 10000)
  })

  channel_user.on("cancel", function(){
    console.log("invite was canceled")
    canceled = true
    document.getElementById("rejectBtn").click()
  })

  let waitingDialog = document.getElementById("
     waitingDialog")
  waitingDialog.addEventListener("close", function(){
    available = true
    waitingDialog.style.display = "none"
  })

  let acceptButton = document.getElementById("acceptBtn")
  acceptButton.addEventListener("click", function(){
    clearTimeout(timeout)
    let payload = {sourceId: userId, destId: inviterId}
    console.log("invite accept")
    channel_user.push("accept", payload)
    .receive("error", e => console.log(e))
    inviteDialog.style.display = "none"
  })

  let rejectButton = document.getElementById("rejectBtn")
  rejectButton.addEventListener("click", function(){
    clearTimeout(timeout)
    let payload = {sourceId: userId, destId: inviterId}
    inviterId = null
    available = true
    if (canceled) {
      canceled = false
```

```
      inviteDialog.style.display = "none"
      return
    }
    console.log("invite reject")
    channel_user.push("reject", payload)
    .receive("error", e => console.log(e))
    inviteDialog.style.display = "none"
  })

  channel_user.join().receive("ok", resp => console.log("
     joined user" + userId, resp))
  .receive("error", reason => console.log("failed user" +
     userId, reason))
```

Ako u kanal `channel_user` pristigne poruka `invite` odbit će se ako igrač već ima poziv na koji nije reagirao. U protivnom otvorit će mu se prozor za pristanak te ima deset sekundi za odgovor. Ako pristigne poruka `cancel`, prozor će se ugasiti. Prihvatom poziva, kanalu će se poslati događaj `accept`, a odbijanjem poslat će se događaj `reject`. Još treba u objekt `Lobby` dodati funkciju `Invite`:

assets/js/lobby.js

```
Invite(userId, id, socket){
  let payload = {sourceId: userId, destId: id}
  let channel_invite = socket.channel("user:" + id)
  channel_invite.join().receive("ok", resp => console.
     log("joined user" + id, resp))
    .receive("error", reason => {
      console.log("failed user" + id, reason)
    })
  channel_invite.push("invite", payload)
    .receive("error", e => {
      console.log(e)
      channel_invite.leave()
    })
  console.log("invited" + id)
  let waitingDialog = document.getElementById("
     waitingDialog")
  waitingDialog.style.display = "block"
  const event = new Event("close")
```

```
    let cancelButton = document.getElementById("cancelBtn"
       )
    cancelButton.addEventListener("click", function(){
      channel_invite.push("cancel", payload)
        .receive("error", e => {
          console.log(e)
          channel_invite.leave()
        })
      waitingDialog.dispatchEvent(event)
    })
    channel_invite.on("reject", resp => {
      if(resp.dest_id != userId || resp.source_id != id)
         return
      waitingDialog.dispatchEvent(event)
      channel_invite.leave()
    })
    channel_invite.on("accept", resp => {
      if(resp.dest_id != userId || resp.source_id != id)
         return
      waitingDialog.dispatchEvent(event)
      channel_invite.leave()
    })
    return channel_invite
}
```

Funkcija se brine za spajanje na kanal da bi se igraču na kojega se stisnulo poslao poziv u igru te prikazuje prozor za čekanje. Čekanje se može prekinuti klikom na gumb za prekidanje. Kad pristigne odgovor iz kanala, kanal će se napustiti.

Sada još treba napraviti novi kontekst za igru, dodati tablicu za igre u bazu, napraviti logiku igre i popratni upravljač i pogled kako bi se korisnici mogli igrati. Nećemo sve raditi ručno kao prije nego ćemo iskoristiti generator da nam olakša posao. To ćemo napraviti naredbom:

```
mix phx.gen.context Games Match matches white_id:references:users
black_id:references:users turn_id:references:users
winner_id:references:users board:string extra_turn:string
```

Dobije se ispis da su izrađene nove datoteke. U naredbi smo naveli ime konteksta, modul koji definira shemu i polja sheme. Naredbom `mix ecto.migrate` dodat će se nova tablica `matches` u bazu. Potrebno je napraviti izmjene u `changeset` funkciji nove sheme:

lib/checkers/games/match.ex

```
    |> cast(attrs, [:board, :extra_turn])
```

Liniju treba zamijeniti s:

lib/checkers/games/match.ex

```
    |> cast(attrs, [:board, :extra_turn, :white_id, :
       black_id, :turn_id, :winner_id])
```

Sad kad postoje igre, trebaju se moći napraviti i zapravo igrati. Funkcijski programi nemaju stanje, ali svejedno je potrebno moći upravljati stanjem. U Elixiru, paralelni procesi i rekurzije obavljaju taj posao. Taj pristup koristi se u Erlang OTP biblioteci u obliku generičkog poslužitelja `GenServer`. Korištenjem generičkog poslužitelja ne moramo se brinuti o implementaciji poslužitelja nego samo napisati kod specifičan za njega. `GenServer` procesi pružaju većinu onoga što se želi od servisa i sami rješavaju probleme koje servisi stvaraju. Zasebni su procesi koji ne dijele ništa s drugim procesima, čime se dobija izolacija koja se i želi postići. Za postavljanje našeg poslužitelja koji će koristiti `GenServer` dovoljno je napraviti novi modul `GameServer` i definirati osnovne funkcije:

lib/checkers/game_server.ex

```
defmodule Checkers.GameServer do
  use GenServer

  def start_link(opts) do
    GenServer.start_link(__MODULE__, opts)
  end

  def init(opts) do
    {:ok, opts}
  end
end
```

Nakon dodavanja u stablo nadzora u `lib/checkers/application.ex`, `GameServer` je spreman za korištenje:

lib/checkers/application.ex

```
children = [
  ...,
  CheckersWeb.GameServer
]
```

`GameServer` popunimo logikom za igru iz datoteke `lib/checkers/game_server.ex` koja je dostupna na [8]. `GameServer` ima funkcije `create_match`, `surrender` i `move`.

`create_match` koristi se za izradu nove igre u bazi. `surrender` će se zvati kad se igrač želi predati. `move` kao argumente prima `match_id`, `user_id`, `old_coord` i `new_coord`. `match_id` i `user_id` potrebni su da se provjeri postoji li igrač u toj igri i je li na redu. Ako je, poslane koordinate moraju biti valjane, tj. na starim koordinatama mora biti njegova figurica koja se može micati, a na novim koordinatama mora biti prazno mjesto na kojega ta figurica može doći valjanim potezom. U ovom trenu, igra se može igrati preko ljuske pozivima funkcija `create_match` i `move`.

Sljedeći korak je korisnicima omogućiti igranje preko preglednika. Za to je potrebna nova ruta u usmjerniku:

lib/checkers_web/router.ex

```
    get "/match/:id", MatchController, :show
```

Nužan je i novi upravljač `MatchController`:

lib/checkers_web/controllers/match_controller.ex

```
defmodule CheckersWeb.MatchController do
  use CheckersWeb, :controller
  plug :authenticate_user when action in [:show]

  alias Checkers.Games

  def show(conn, %{"id" => id}) do
    user_id = get_session(conn, :user_id)
    match = Games.get_match!(id)
    if match.winner_id == nil && match.turn_id != nil do
      render(conn, "show.html", match: match, user_id:
         user_id)
    else
      redirect(conn, to: "/lobby")
    end
  end
end
```

Upravljač će preko id-a meča iz baze dohvatiti podatke o meču. Ako igra nije gotova, pokazat će se trenutno stanje igre i moći će se dalje igrati. Ako je igra gotova, preusmjerit će se na lobby. Upravljaču treba i pripadni pogled `MatchView`:

lib/checkers_web/views/match_view.ex

```
defmodule CheckersWeb.MatchView do
  use CheckersWeb, :view
```

```
end
```

Pogled treba i predložak `lib/checkers_web/templates/match/show.html.eex` koji je dostupan na [8]. Predložak se sastoji od tablice koja će se popuniti figuricama na odgovarajućim mjestima. Ispod tablice ispisivat će se koji igrač je na redu, a postoji i gumb za predaju. Dodajmo još i CSS datoteku `assets/css/match.css` koja je dostupna na [8] te ju uvezimo u glavnu CSS datoteku:

assets/css/app.css

```
@import "./match.css";
```

Sve slike koje ćemo koristiti stavljaju se u mapu `assets/static/images`, a dostupne su na [8]. Želimo da korisnika preusmjeri na igru kad prihvati poziv za nju, no prije toga novu igru treba napraviti. U tu svrhu u kanal `UserChannel` treba dodati sljedeće:

lib/checkers_web/channels/user_channel.ex

```
  alias Checkers.GameServer

  def handle_in("accept", params, socket) do
    result = GameServer.create_match(params["destId"],
       params["sourceId"])
    case result do
      {:ok, match} ->
        broadcast!(socket, "accept", %{
        source_id: params["sourceId"],
        dest_id: params["destId"],
        match_id: match.id
        })
        {:reply, :ok, socket}
      _ ->
        {:noreply, socket}
    end
  end
```

Za samo preusmjeravanje nakon prihvaćanja poziva, u datoteci `assets/js/lobby.js` na kraj funkcije onReady treba dodati:

assets/js/lobby.js

```
channel_user.on("accept", resp => {
  if(resp.source_id == userId) {
    window.location.href = "/match/" + resp.match_id
```

```
  }
})
```

I u istoj datoteci u funkciji `Invite` nakon linija:

assets/js/lobby.js

```
        channel_invite.on("accept", resp => {
            if(resp.dest_id != userId || resp.source_id
               != id) return
```

treba dodati liniju:

assets/js/lobby.js

```
            window.location.href = "/match/" + resp.
               match_id
```

Za komunikaciju na stranici gdje će se igrati igra potreban je novi kanal `MatchChannel` pa ga dodajmo u `UserSocket`:

lib/checkers_web/channels/user_socket.ex

```
channel "match:*", CheckersWeb.MatchChannel
```

`MatchChannel` treba i izraditi:

lib/checkers_web/channels/match_channel.ex

```
defmodule CheckersWeb.MatchChannel do
  use CheckersWeb, :channel

  def join("match:" <> match_id, _params, socket) do
    {:ok, assign(socket, :match_id, String.to_integer(
       match_id))}
  end

  alias Checkers.Accounts

  def handle_in(event, params, socket) do
    user = Accounts.get_user!(socket.assigns.user_id)
    handle_in(event, params, user, socket)
  end

  alias Checkers.GameServer
```

```
def handle_in("surrender", params, user, socket) do
  result = GameServer.surrender(params["matchId"],
     user.id)
  case result do
    {:ok, match} ->
      broadcast!(socket, "move", %{
        board_string: match.board,
        turn_id: match.turn_id,
        extra_turn: match.extra_turn,
        winner_id: match.winner_id,
        white_id: match.white_id,
        black_id: match.black_id
      })
      {:reply, :ok, socket}

    _->
      {:noreply, socket}
  end
end

def handle_in("move", params, user, socket) do
  old_coord = %{:x => params["oldCoord"]["x"], :y =>
     params["oldCoord"]["y"]}
  new_coord = %{:x => params["newCoord"]["x"], :y =>
     params["newCoord"]["y"]}
  result = GameServer.move(params["matchId"], user.id,
      old_coord, new_coord)
  case result do
    {:ok, match} ->
      broadcast!(socket, "move", %{
        board_string: match.board,
        turn_id: match.turn_id,
        extra_turn: match.extra_turn,
        winner_id: match.winner_id,
        white_id: match.white_id,
        black_id: match.black_id
      })
      {:reply, :ok, socket}
```

```
      _ ->
        {:noreply, socket}
      end
  end
end
```

Kanal će se brinuti da na primljeni događaj `move` poslužitelju javi koji potez treba napraviti ili da na događaj `surrender` poslužitelju javi da se igrač predao. Za prikaz i samu interakciju s igrom bit će zadužena datoteka `assets/js/match.js` koja je dostupna na [8]. Uvezimo ju u glavnu JS datoteku:

assets/js/app.js

```
import Match from "./match"
Match.init(socket, document.getElementById("match"))
```

`Match.js` će u svakom trenutku crtati trenutno stanje ploče. Igrač koji je na redu može kliknuti na svoju figuricu nakon čega mu se prikazuje na koja polja ta figurica može doći. Valjanim potezom šalju se podaci kanalu te se crta novo stanje ploče. Korisniku želimo omogućiti da vidi i trenutne mečeve u kojima se nalazi te da vidi rezultate zadnjih nekoliko mečeva. Zato u kontekst `Games` dodajemo sljedeće:

lib/checkers/games.ex

```
  def current_matches(id) do
    Repo.all(from(m in Match, where: (m.white_id == ^id
       or m.black_id == ^id) and is_nil(m.winner_id)))
  end

  def last_matches(id) do
    Repo.all(from(m in Match, where: (m.white_id == ^id
       or m.black_id == ^id) and (m.winner_id == m.
       white_id or m.winner_id == m.black_id), order_by:
        [desc: :updated_at],limit: 10))
  end
```

Funkcije pomoću Ecto upita dohvaćaju trenutne kao i zadnjih deset mečeva. U upravljaču `LobbyController` potrebno je akciju `show` izmijeniti na sljedeći način:

lib/checkers_web/controllers/lobby_controller.ex

```
  alias Checkers.Games
  def show(conn, _) do
```

```
    user_id = get_session(conn, :user_id)
    current_matches = Games.current_matches(user_id)
    render(conn, "show.html", user_id: user_id,
       current_matches: current_matches)
  end
```

Samo smo dodali da upravljač prosljeđuje podatke o trenutnim mečevima. U pogled `LobbyView` za izvlačenje podataka o trenutnim mečevima dodajemo sljedeće:

lib/checkers_web/views/lobby_view.ex

```
alias Checkers.Accounts
def get_details(current_matches) do
  Enum.reduce(current_matches, [], fn x, acc ->
    white = Accounts.get_user!(x.white_id)
    black = Accounts.get_user!(x.black_id)
    acc ++ [{x.id, white.username, black.username}]
    # {key + 1, Map.put(acc, key, {x.id, x.white_id, x
       .black_id})}
  end)
end
```

Izvučene podatke prikazujemo u predlošku dodavanjem novog elementa:

lib/checkers_web/templates/lobby/show.html.eex

```
<div class="col">
    <h3>Your current matches</h3>
    <div id="current-matches">
        <ul>
            <%= for {id, white, black} <-
               get_details(@current_matches) do %>
                <li><%= link white <> " vs " <>
                   black, to: Routes.match_path(
                   @conn, :show, id) %></li>
            <% end %>
        </ul>
    </div>
</div>
```

Korisnik u ovom trenu u lobbyju može vidjeti svoje trenutne mečeve. Preostaje samo prikaz prošlih mečeva. Za to je potrebno u upravljaču `UserController` akciju `show` zamijeniti sljedećim kodom:

lib/checkers_web/controllers/user_controller.ex

```
  alias Checkers.Games
  def show(conn, %{"id" => id}) do
    user = Accounts.get_user(id)
    last_matches = Games.last_matches(id)
    render(conn, "show.html", user: user, last_matches:
       last_matches)
  end
```

Upravljač prosljeđuje podatke o posljednjim mečevima pogledu. U pogledu dodajemo funkciju koja izvlači proslijeđene podatke:

lib/checkers_web/views/user_view.ex

```
  alias Checkers.Accounts
  def get_details(last_matches) do
    Enum.reduce last_matches, [], fn x, acc ->
      white = Accounts.get_user!(x.white_id)
      black = Accounts.get_user!(x.black_id)
      winner =
        if x.winner_id == x.white_id do
          white
        else
          black
        end
      acc ++ [{white.username, black.username, winner.
         username, x.white_id, x.black_id, x.winner_id}]
    end
  end
```

Da bi se proslijeđeni podaci prikazali, sadržaj predloška zamijenimo sljedećim:

lib/checkers_web/templates/user/show.html.eex

```
<h3><%= @user.username %>'s recent matches</h1>
<div id="last-matches">
 <table>
  <tbody>
   <tr>
    <th>White <img src="<%= Routes.static_path(@conn, "/
       images/white_pawn.png")%>" alt="white"></th>
    <th>Black <img src="<%= Routes.static_path(@conn, "/
       images/black_pawn.png")%>" alt="black"></th>
```

```
      <th>Winner <img src="<%= Routes.static_path(@conn, "
         /images/winner.png")%>" alt="winner"></th>
    </tr>
    <%= for {white, black, winner, white_id, black_id,
       winner_id} <- get_details(@last_matches) do %>
    <tr> <td> <%= link white, to: Routes.user_path(@conn,
        :show, white_id) %> </td> <td> <%= link black, to
       : Routes.user_path(@conn, :show, black_id) %> </td
       > <td> <%= link winner, to: Routes.user_path(@conn
       , :show, winner_id) %> </td> </tr>
    <% end %>
   </tbody>
  </table>
</div>
```

I za kraj ćemo samo još u novu datoteku `assets/css/show_user.css` staviti sljedeće:

assets/css/show_user.css

```
#last-matches img {
    width: 20;
    height: 20px;
}
```

te ju uvesti u glavnu CSS datoteku:

assets/css/app.css

```
@import "./show_user.css";
```

Bibliografija

[1] *Web stranice PostgreSQL-a*, https://www.postgresql.org/download/, 1996, posjećeno u kolovozu 2020.

[2] *Web stranice Ecto dokumentacije*, https://hexdocs.pm/ecto/Ecto.html, 2012, posjećeno u kolovozu 2020.

[3] *Web stranice Elixira*, https://elixir-lang.org/, 2012, posjećeno u kolovozu 2020.

[4] *Web stranice Elixirove dokumentacije*, https://hexdocs.pm/elixir/Kernel.html, 2012, posjećeno u kolovozu 2020.

[5] *Web stranice Phoenix dokumentacije*, https://hexdocs.pm/phoenix/Phoenix.html, 2012, posjećeno u kolovozu 2020.

[6] *Github stranica dwyl-a*, https://github.com/dwyl/learn-elixir, 2016, posjećeno u kolovozu 2020.

[7] *Web stranice Elixir School-a*, https://elixirschool.com/en/, 2018, posjećeno u kolovozu 2020.

[8] *Aplikacija izrađena za potrebe diplomskog rada*, https://github.com/teskeras/checkers, 2020, posjećeno u kolovozu 2020.

[9] *Upute za instalaciju inotify*, https://hexdocs.pm/phoenix/installation.html#inotify-tools-for-linux-users, 2020, posjećeno u kolovozu 2020.

[10] *Web stranice Node.js-a*, https://nodejs.org/en/, 2020, posjećeno u kolovozu 2020.

[11] *Web stranice Phoenix okruženja*, https://www.phoenixframework.org/, 2020, posjećeno u kolovozu 2020.

[12] Ulisses Almeida, *Learn Functional Programming with Elixir*, The Pragmatic Bookshelf, 2018.

[13] José Valim Chris McCord, Bruce Tate, *Programming Phoenix*, The Pragmatic Programmers, 2016.

[14] Lance Halvorsen, *Functional Web Development with Elixir, OTP, and Phoenix*, The Pragmatic Programmers, 2018.

Sažetak

U ovom radu prezentirani su programski jezik Elixir i okruženje Phoenix. Elixir je funkcijski programski jezik poznat po brzini i produktivnosti. Phoenix je MVC okruženje za razvoj web aplikacija i napisan je u Elixiru.

U prvom poglavlju opisali smo Elixir. Na samom početku objašnjeno je kako ga instalirati i koristiti. Naveli smo kako koristiti varijable i funkcije. Opisali smo provođenje kontrole toka. Objasnili smo što su rekurzivne funkcije. Pojasnili smo što su to funkcije višeg reda. Na kraju poglavlja dotakli smo se nečistih funkcija.

U drugom poglavlju opisan je Phoenix. Najprije je opisano kako instalirati Phoenix i koristiti ga. Građenjem aplikacije upoznajemo se s osnovnim dijelovima kao što su upravljač i pogled. Objasnili smo što je Ecto i kako se upravlja bazom podataka. Naveli smo kako provesti autentifikaciju korisnika. Na kraju smo objasnili što su kanali i kako preko njih komunicirati.

Summary

In this thesis, Elixir programming language and Phoenix framework are presented. Elixir is a functional programming language famous for his speed and productivity. Phoenix is an MVC framework for web development written in Elixir.

In the first chapter, we described Elixir. In the beginning, it was explained how to install and use it. We stated how to use variables and functions. We described the control flow. We explained what recursive functions are. We clarified what higher-order functions are. In the end, we touched upon impure functions.

In the second chapter, Phoenix was described. Firstly, it was described how to install Phoenix and use it. Building an application familiarizes ourselves with basic parts like controller and view. We explained what Ecto is and how to manage a database. We stated how to perform user authentication. In the end, we explained what channels are and how to communicate through them.

Životopis

Rođen sam 14. srpnja 1993. godine u Zagrebu. Nakon završene Osnovne škole Jurja Habdelića 2008. godine upisujem Gimnaziju Velika Gorica u Velikoj Gorici. Završio sam sa školovanjem 2012. godine te tada upisujem preddiplomski studij matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu. Stjecanjem titule sveučilišnog prvostupnika matematike 2017. godine završavam preddiplomski studij te iste godine upisujem diplomski studij Računarstva i matematike, također na Prirodoslovno-matematičkom fakultetu u Zagrebu.