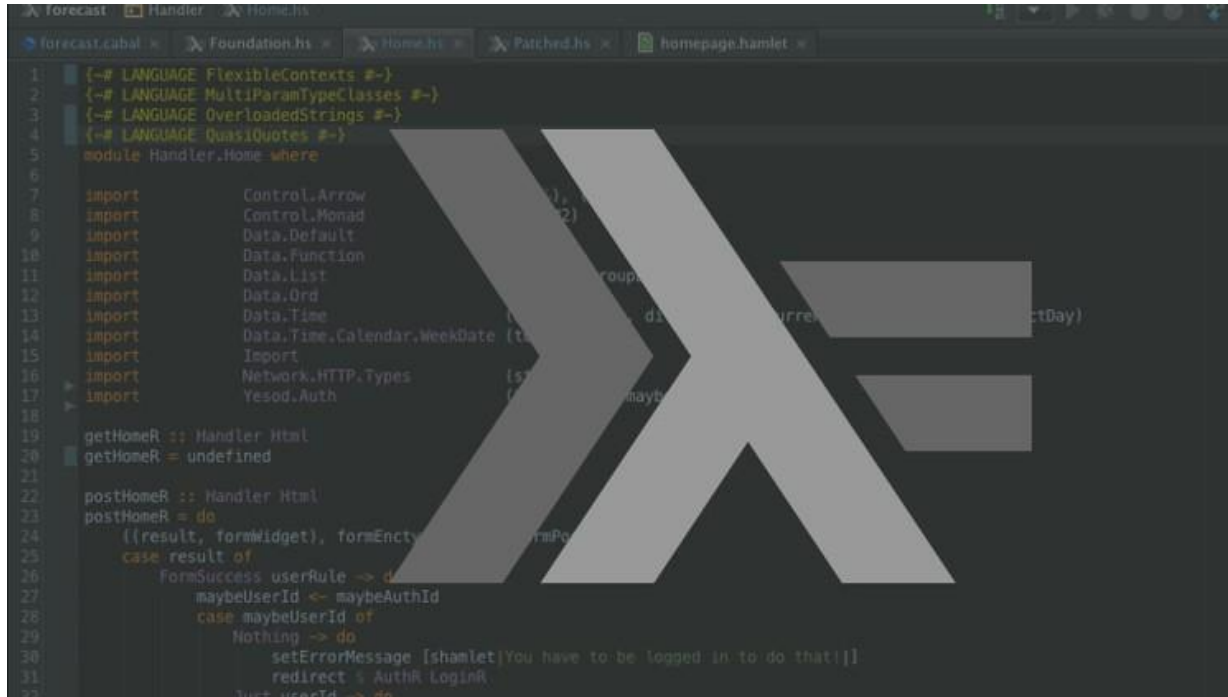


Implementation of fmap



```
1 {-# LANGUAGE FlexibleContexts #-}
2 {-# LANGUAGE MultiParamTypeClasses #-}
3 {-# LANGUAGE OverloadedStrings #-}
4 {-# LANGUAGE QuasiQuotes #-}
5 module Handler.Home where
6
7 import Control.Arrow
8 import Control.Monad
9 import Data.Default
10 import Data.Function
11 import Data.List
12 import Data.Ord
13 import Data.Time
14 import Data.Time.Calendar.WeekDate
15 import Import
16 import Network.HTTP.Types
17 import Yesod.Auth
18
19 getHomeR :: Handler Html
20 getHomeR = undefined
21
22 postHomeR :: Handler Html
23 postHomeR = do
24   ((result, formWidget), formEnctype) <- runPost
25   case result of
26     FormSuccess userRule -> do
27       maybeUserId <- maybeAuthId
28       case maybeUserId of
29         Nothing -> do
30           setErrorMessage [shamlet|You have to be logged in to do that!|]
31           redirect % AuthR.LoginR
32         Just userId -> do
```

Functors



We already know how to apply functions:

```
λ> (+3) 2      ➡ 5
```

But...

```
λ> (+3) (Just 2)
```



In this case, we can use `fmap`!

```
λ> fmap (+3) (Just 2)      ➡ Just 5
λ> fmap (+3) Nothing       ➡ Nothing
```

And it also works with `Either`, lists, tuples and functions:

```
λ> fmap (+3) (Right 2)      ➡ Right 5
λ> fmap (+3) (Left "error") ➡ Left "error"

λ> fmap (+3) [1, 2, 3]      ➡ [4, 5, 6]      -- same as map
λ> fmap (+3) (1, 6)         ➡ (1, 9)         -- because (,) is a type
λ> (fmap (*2) (+1)) 3       ➡ 8              -- same as (.)
```

Implementation of fmap



`fmap` applies a function to the elements of a generic container `f` `a` returning a container of the same type.

`fmap` is a function of the instances of the class `Functor`:

```
λ> :type fmap
fmap :: Functor f => (a -> b) -> (f a -> f b)
```

Where

```
λ> :info Functor
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```