Fortgeschrittene Funktionale Programmierung

Teil 9: Webapplikationen mit Yesod

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

28. Januar 2019

1. Grundlagen
2. Shakespeare
   - Hamlet
   - Lucius
   - Cassius
   - Julius
3. Yesod
   - Widgets
   - Foundation Type
   - Routing
   - Handling
   - Formulare
   - Sessions
     - Messages
     - Ultimate Destination
4. Persistent
   - Spezifizieren
   - Migration
   - Zugriff
   - Integration in Yesod
5. Ausblick
   - Esqueleto

Web-App Frameworks für Haskell

Zur Entwicklung von **Webapplikation** bietet sich die Verwendung eines Frameworks an, welches alle grundlegenden Funktionalitäten durch Bibliotheken und vordefinierte Programmgerüste bereitgestellt.

Die wichtigsten Web Development Frameworks für Haskell sind:
Happstack, Yesod, Snap and Servant — mix & match möglich!

Im folgenden betrachten wir Yesod — `>=1.4 && <=1.6`

- Typ-sichere URLs — generell viele statische Prüfungen
- Templating / DSLs — viele modulare Einzelteile
- integrierte Datenbankanbindung — `persist / conduit`
- REST Architektur — REpresentational State Transfer
  zustandslos, d.h. gleiche URL = gleiche Webseite

Yesod Installation

Installation mit `stack` wird wegen vieler Abhängigkeiten dringend empfohlen:

```
> stack templates
> stack new mein-projekt yesod-sqlite
> cd mein-projekt
> stack build yesod-bin --install-ghc
> stack build
> stack exec -- yesod devel
```

Folgende Webseite sollte dann *lokal* abrufbar sein:
http://localhost:3000/

Siehe auch https://www.yesodweb.com/page/quickstart

Yesod Installation

Installation mit `stack` wird wegen vieler Abhängigkeiten dringend empfohlen:

```
> stack templates
> stack new mein-projekt yesodweb/sqlite
> cd mein-projekt
> stack build yesod-bin --install-ghc
> stack build
> stack exec -- yesod devel
```

Folgende Webseite sollte dann *lokal* abrufbar sein:
http://localhost:3000/

Siehe auch https://www.yesodweb.com/page/quickstart

Scaffolding Tool

Stack bringt Templates (Grundgerüste) mit:

```
stack new my-project yesod-sqlite
```

Es gibt weitere Templates, anzeigen mit: `stack templates`

Gerüste nutzen viele nicht-zwingende, aber sinnvolle Voreinstellung, z.B. sind viele Dinge in mehreren Dateien getrennt, was nicht unbedingt notwendig ist. Vorlesungs-Beispiele oft in einer Datei.

Development Webserver auf `http://localhost:3000` starten:

```
stack exec -- yesod devel
```

Eventuell Umgebungsvariablen `HOST` und `PORT` einstellen

Nicht von allen Templates unterstützt

- `yesod devel` überwacht den Quellcode; ggf. wird der Development Webserver kompiliert & neugestartet.
- Compiler-Optionen, Paket-Abhängigkeiten in `package.yaml` einstellen Konfigurationsdateien werden automatisch erstellt

Hello Yesod

```haskell
{-# LANGUAGE
  OverloadedStrings,
  TypeFamilies,
  TemplateHaskell,
  QuasiQuotes
#-}

module Main where

import Yesod


data MyApp = MkApp
instance Yesod MyApp
```

```haskell
mkYesod "MyApp" [parseRoutes|
  / HomeR GET
|]

getHomeR :: Handler Html
getHomeR = defaultLayout $ do
  setTitle "HelloWorld"
  toWidget [whamlet|
      <h2>Hello Yesod!
      Some text that
      is <i>displayed</i> here.
    |]

main :: IO ()
main = warp 3000 MkApp
```

Hello Yesod

```haskell
{-# LANGUAGE
  OverloadedStrings,
  TypeFamilies,
  TemplateHaskell,
  QuasiQuotes
#-}

module Main where

import Yesod


data MyApp = MkApp
instance Yesod MyApp
```

```haskell
mkYesod "MyApp" [parseRoutes|
  / HomeR GET
|]

getHomeR :: Handler Html
getHomeR = defaultLayout $ do
  setTitle "HelloWorld"
  toWidget [whamlet|
      <h2>Hello Yesod!
      Some text that
      is <i>displayed</i> here.
    |]

main :: IO ()
main = warp 3000 MkApp
```

Shakespearean Templates

```
[hamlet|
  <h2>Hello Yesod!
  Some text that is <i>displayed</i> here.
|]
```

HTML, CSS und JavaScript werden in Yesod mit den Mechanismen für Template Haskell programmiert/manipuliert. Dazu werden spezialisierte Quasi-Quoter `[|...|]` bereitgestellt:

Sprache	Quasiquoter	Dateiendung
HTML	`hamlet`	.hamlet
CSS	`cassius`	.cassius
CSS	`lucius`	.lucius
JavaScript	`julius`	.julius
I18n interpolierter Text	`stext/ltext`	

Spezialisierte Varianten `shamlet`, `whamlet`, ... unterscheiden sich meist nur durch Ergebnistyp.

Yesod Templates und Interpolation

Entsprechend dem Spleißen für Haskell-Code werden Templates durch **Interpolation** eingebunden:

- `#{ }` Interpolation für Variablen im Scope (escaped)
- `@{ }` Typsichere URL Interpolation, also `@{HomeR}`
- `^{ }` Template embedding, fügt Template gleichen Typs ein
- `_{ }` Internationalisierter Text, fügt Übersetzung ein

Damit können wir Templates dynamisch gestalten:

```
[whamlet|
  Value of fib22 is #{show (fib 22)}
|]
```

Ergebnistyp einer Interpolation muss immer eine Instanz der Typklasse `ToHtml` sein — funktioniert ganz analog zu `Show`

Fallstrick: Typ von interpolierten Ausdrücken muss inferierbar sein! — ggf. Typ angegeben

Variablen Interpolation

Interpolation von URLs funktioniert ähnlich:

```
let foo = show (fib 22) in
    [whamlet|
      Value of foo is #{foo}

      Return to <a href=@{Home}>Homepage
      .
    |]
```

Dabei muss `Home` ein Konstruktor/Wert des Datentyps für das *Routing* dieser Webanwendung sein, welches wir später genauer betrachten werden.

Hamlet I

Hamlet funktioniert wie gewöhnliches HTML plus Interpolation

Zusätzlich gilt:

- Schließende HTML-Tags werden durch Einrücken ersetzt:

```
<p>Some paragraph.
  <ul><li>Item 1</li>
      <li>Item 2</li>
  </ul></p>
<p>Next paragraph.</p>
```

dieses HTML wird in Hamlet so geschrieben:

```
<p>Some paragraph.
  <ul>
    <li>Item 1
    <li>Item 2
<p>Some paragraph.
```

Quasiquoter generiert oberen Code aus dem unteren.

Hamlet II

Zusätzlich gilt:

- Kurze geschlossene inline Tags sind zulässig:

```
<p>Some <i>italic</i> paragraph.
```

  Wichtig: Ein Tag am Zeilenanfang wird immer nur durch Einrückung geschlossen.

- Leerzeichen vor und nach Tags und an Zeilenanfang und -ende werden entfernt. An diesen Positionen explizit gewollte Leerzeichen müssen mit # und \ markiert werden:

```
<p>
  Some #
  <i>italic
  \ paragraph.
```

Hamlet III

Zusätzlich gilt:

- Attribute funktionieren wie in HTML, d.h. Gleichheitszeichen, Wert und Anführungszeichen sind meist optional.

  Abkürzungen für IDs, Klassen und Konditionale erlaubt:

  ```
  <p #paragraphid .class1 .class2>
  <p :someBool:style="color:red">
  <input type=checkbox :isChecked:checked>
  ```

- Ein Attribut-Paar `attr::(Text,Text)` oder mehrere `attrs::[(Text,Text)]` können auch direkt eingebunden werden:

  ```
  <p *{attrs}>
  ```

Hamlet IV

Hamlet erlaubt auch logische Konstrukte

- Konditional

```
$if isAdmin
  <p>Hallo mein Administrator!
$elseif isLoggedIn
  <p>Du bist nicht mein Administrator.
$else
  <p>Wer bist Du?
```

- Einfache Schleifen:

```
$if null people
  <p>Niemand registriert.
$else
  <ul>
    $forall person <- people
      <li>#{show person}
```

Hamlet V

- Maybe

```
$maybe name <- maybeName
  <p>Dein Name ist #{name}
$nothing
  <p>Ich kenne Dich nicht.
```

  Pattern-Matching und unvollständige Fälle sind erlaubt:

```
$maybe Person vorname nachname <- maybePerson
  <p> Dein Name ist #{vorname} #{nachname}
```

- Volles Pattern-Matching mit Case

```
$case foo
  $of Left bar
      <p>Dies war links: #{bar}
  $of Right baz
      <p>Dies war rechts: #{baz}
```

HAMLET VI

- `$with` ist das neue `let`, also für lokale Definitionen

```
$with foo <- myfun argument $ otherfun more args
  <p>
    Einmal ausgewertetes foo hier #{foo}
    und da #{foo} und dort #{foo} verwendet.
```

- Abkürzungen für Standard Komponenten vordefiniert:

```
$doctype 5
```

steht zum Beispiel für

```
<!DOCTYPE html>
```

Lucius

Lucius akzeptiert ganz normales CSS. Zusätzlich ist erlaubt

- Interpolation für Variablen `#{}`, URLs `@{}` und Mixins `^{}`
- CSS Blöcke dürfen verschachtelt werden
- Es können lokale Variablen deklariert werden

Beispiel:

```
article code { background-color: grey; }
article p { text-indent: 2em; }
article a { text-decoration: none; }
```

kann bei Bedarf umgeschrieben werden zu

```
@backgroundcolor: grey;
article {
  code { background-color: #{backgroundcolor}; }
  p { text-indent: 2em; }
  a { text-decoration: none; } }
```

Beispiel: Lucius Mixin

```
transition val = -- ist kein korrektes Lucius...
   [luciusMixin|
       -webkit-transition: #{val};
       -moz-transition: #{val};
       -ms-transition: #{val};
       -o-transition: #{val};
       transition: #{val};
   |]
 myCSS = -- ...aber korrekt innerhalb Lucius Klasse
   [lucius|
       .some-class {
           ^{transition "all 4s ease"}
       }
   |]
```

Cassius

Eignet sich für Whitespace-sensitive Haskell-Programmierer als Alternative zu Lucius:

Cassius wird zu Lucius übersetzt. Klammern und Semikolon *müssen* immer durch Einrücken ersetzt werden:

```
#banner
  border: 1px solid #{bannerColor}
  background-image: url(@{BannerImageR})
```

⇒ Für vorhandenen CSS Code immer Lucius einsetzen
(auch bei Verwendung von Front-End Frameworks, z.B. Bootstrap)

⇒ Für neuen CSS Code das bequemere Cassius einsetzen

Nach Interpolation ist mischen problemlos möglich.

Julius

Julius akzeptiert gewöhnliches JavaScript, plus

- `#{}` Variablen Interpolation
- `@{}` URL Interpolation
- `^{}` Template Embedding von anderen JavaScript Templates

Sonst ändert sich nichts, auch nicht an Einrückungen!

Quasiquoter für Varianten wie CoffeScript sind auch verfügbar

Scaffolding minimiert JavaScript vor Auslieferung `hjsmin`

⇒ Es gibt inzwischen auch mehrere Ansätze, JavaScript direkt aus Haskell zu generieren, siehe z.B. Haste oder GhcJs.

Widgets

Widgets fassen einzelne Templates von verschiedenen Shakespeare-Sprachen zu einer Einheit zusammen:

```
getRootR = defaultLayout $ do
  setTitle "My Page Title"
  toWidget [lucius| h1 { color: green; } |]
  addScriptRemote "https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/j
  toWidget [julius|
    $(function() {
      $("h1").click(function(){ alert("Clicked the heading!"); });
    });
  |]
  toWidgetHead [hamlet| <meta name=keywords content="keywords">|]
  toWidget [hamlet| <h1>Here's one way for including content |]
  [whamlet| <h2>Here's another |]
  toWidgetBody [julius| alert("This is included in the body"); |]
```

Widget-Monade erlaubt kombinieren dieser Bausteine; alles wird automatisch dahin sortiert, wo es hingehört.

Widgets – whamlet

Template embedding erlaubt normalerweise nur die Einbettung aus der gleichen Template Sprache. Dagegen erlauben `whamlet` bzw. `.whamlet`-Dateien die Einbettung von Widgets in Hamlet:

```
page = [whamlet|
    <p>This is my page. I hope you enjoyed it.
    ^{footer}
  |]

footer = do
    toWidget [lucius| footer { font-weight: bold;
                               text-align: center } |]
    toWidget [hamlet|
        <footer>
          <p>That's all folks!
      |]
```

Frische Namen für frische Widgets

Bei der Kombination von Widgets könnten Namenskonflikte auftreten. Dies wird durch dynamische IDs verhindert werden:

```
getRootR = defaultLayout $ do
  headerClass <- lift newIdent
  toWidget [hamlet|<h1 .#{headerClass}>My Header|]
  toWidget [lucius| .#{headerClass} { color: green; } |]
```

Die Funktion `newIdent` erlaubt die Erzeugung von frischen IDs.

Wir müssen `lift` einsetzen, da `newIdent` nicht in der `Widget`-Monade, sondern in der `Handler`-Monade ausgeführt wird, welche ineinander verschachtelt sind.

Hello Yesod

```
{-# LANGUAGE
  OverloadedStrings,
  TypeFamilies,
  TemplateHaskell,
  QuasiQuotes
#-}

module Main where

import Yesod


data MyApp = MkApp
instance Yesod MyApp
```

```
mkYesod "MyApp" [parseRoutes|
  / HomeR GET
|]

getHomeR :: Handler Html
getHomeR = defaultLayout $ do
  setTitle "HelloWorld"
  toWidget [whamlet|
      <h2>Hello Yesod!
      Some text that
      is <i>displayed</i> here.
    |]

main :: IO ()
main = warp 3000 MkApp
```

Hello Yesod

```
{-# LANGUAGE
  OverloadedStrings,
  TypeFamilies,
  TemplateHaskell,
  QuasiQuotes
#-}

module Main where

import Yesod
```

```
data MyApp = MkApp
instance Yesod MyApp
```

```
mkYesod "MyApp" [parseRoutes|
  / HomeR GET
|]

getHomeR :: Handler Html
getHomeR = defaultLayout $ do
  setTitle "HelloWorld"
  toWidget [whamlet|
      <h2>Hello Yesod!
      Some text that
      is <i>displayed</i> here.
    |]

main :: IO ()
main = warp 3000 MkApp
```

Yesod Typklasse

Einstellungen einer Yesod Applikation werden mit Hilfe eines **Foundation** Datentyps vorgenommen.

Der Wert muss dazu nichts direkt speichern; Einstellungen werden über die Instanzdeklaration zur `Yesod`-Typklassen vorgenommen.

Diese Klasse fasst alle möglichen Einstellungen zusammen:

- Rendern und parsen von URLs
- Funktion `defaultLayout`
- Authentifizierung
- Sitzungsdauer
- Cookie Handling
- Fehlerbehandlung und Aussehen der Fehlerseiten
- Externe CSS, Skripte und statische Dateien
- ...

Beispiel: eigenes Error-Handling

Für alle Einstellungen gibt es vordefinierte Standards, welche bei Bedarf überschrieben werden können:

```
data MyWebApp = MkWebApp        -- Foundation Type

instance Yesod MyWebApp         -- Defaults
```

- Default-Definition `errorHandler = defaultErrorHandler` jetzt geändert für den Fall `NotFound`.
- `myNotFoundHandler` ist selbst geschriebener Handler
- Überschreiben von `defaultLayout` verändert bereits auch Aussehen von `defaultErrorHandler`

Beispiel: eigenes Error-Handling

Für alle Einstellungen gibt es vordefinierte Standards, welche bei Bedarf überschrieben werden können:

```
data MyWebApp = MkWebApp       -- Foundation Type

instance Yesod MyWebApp where
  errorHandler NotFound = myNotFoundHandler
  errorHandler other    = defaultErrorHandler other
```

- Default-Definition `errorHandler = defaultErrorHandler` jetzt geändert für den Fall `NotFound`.
- `myNotFoundHandler` ist selbst geschriebener Handler
- Überschreiben von `defaultLayout` verändert bereits auch Aussehen von `defaultErrorHandler`

Beispiel: Parameter

Der Foundation Type kann auch Parameter tragen. Innerhalb der **Handler** Monade kann man diese mit `getYesod` wieder auslesen:

```
data MyWebApp = MkWebApp { intPar :: Int
                         , varPar :: TVar String }

myHandler :: Handler Html
myHandler = do
  master <- getYesod                          -- master :: MyWebApp
  let myint = intPar  master                  -- myint  :: Int
  mystr <- readTVarIO $ varPar master -- mystr  :: String
```

- Sammelt alle statischen Parameter an einer Stellen
- `MVar`/`TVar` ermöglichen gemeinsamen Zustand für Handler

kein `IORef`

Hello Yesod

```
{-# LANGUAGE
  OverloadedStrings,
  TypeFamilies,
  TemplateHaskell,
  QuasiQuotes
#-}

module Main where

import Yesod


data MyApp = MkApp
instance Yesod MyApp
```

```
mkYesod "MyApp" [parseRoutes|
  / HomeR GET
|]

getHomeR :: Handler Html
getHomeR = defaultLayout $ do
  setTitle "HelloWorld"
  toWidget [whamlet|
      <h2>Hello Yesod!
      Some text that
      is <i>displayed</i> here.
    |]

main :: IO ()
main = warp 3000 MkApp
```

Hello Yesod

```haskell
{-# LANGUAGE
  OverloadedStrings,
  TypeFamilies,
  TemplateHaskell,
  QuasiQuotes
#-}

module Main where

import Yesod


data MyApp = MkApp
instance Yesod MyApp
```

```haskell
mkYesod "MyApp" [parseRoutes|
  / HomeR GET
|]
```

```haskell
getHomeR :: Handler Html
getHomeR = defaultLayout $ do
  setTitle "HelloWorld"
  toWidget [whamlet|
      <h2>Hello Yesod!
      Some text that
      is <i>displayed</i> here.
    |]

main :: IO ()
main = warp 3000 MkApp
```

Routing & Handling

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                 RootR     GET
  /blog/help        BlogHelpR GET
  /blog/#Int        BlogPostR GET POST
  /wiki/*WikiPfad   WikiR
  /static           StaticR   Static getStatic
|]
```

Definiert die **gesamte Sitemap** der Webapplikation.
Ausnahme: Kombination mit Yesod Unter-Websites   hier: `Static`

DSL wird innerhalb des Quasiquoters `parseRoutes` angegeben, oder in einer separaten Datei

Beim Spleißen können View-Patterns entstehen, weshalb Spracherweiterung `ViewPatterns` aktiviert sein muss

ROUTING

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                  RootR     GET
  /blog/help         BlogHelpR GET
  /blog/#Int         BlogPostR GET POST
  /wiki/*WikiPfad    WikiR
  /static            StaticR   Static getStatic
|]
```

Zuerst wird der Pfad angegeben. Es gibt drei Arten von Pfaden:

1. Statische Pfade, z.B. /blog/help
2. Dynamische Pfade *enthalten* (mehrere) /#<Typ> Fragmente, wobei <Typ> Instanzen für PathPiece,Eq,Read,Show haben muss
3. Dynamische Multipfade *enden* mit /*<Typ>, wobei <Typ> Instanzen für PathMultiPiece,Eq,Read,Show haben muss
   Es darf auch /+<Typ> geschrieben werden, z.B. wegen CPP

PathPiece & PathMultiPiece

Klassen aus Modul `Yesod.Dispatch` legen lediglich Parser fest:

```
class PathPiece s where
  fromPathPiece :: Text -> Maybe s
  toPathPiece   :: s -> Text

class PathMultiPiece s where
  fromPathMultiPiece :: [Text] -> Maybe s
  toPathMultiPiece   :: s -> [Text]
```

`PathPiece`-Instanzen für `Int`, `Integer`, `String`, `Text` und `PathMultiPiece`-Instanzen für `[String]` und `[Text]` vordefiniert

Fallstrick: `read` kann Ausnahme werfen! Meist reicht schon

```
maybeRead :: Read a => String -> Maybe a
maybeRead (reads -> [(x,"")]) = Just x
maybeRead                _    = Nothing
```

siehe auch `readMay` aus Classy-Prelude

PathPiece & PathMultiPiece

Klassen aus Modul Yesod.Dispatch legen lediglich Parser fest:

```
class PathPiece s where
  fromPathPiece :: Text -> Maybe s
  toPathPiece   :: s -> Text

class PathMultiPiece s where
  fromPathMultiPiece :: [Text] -> Maybe s
  toPathMultiPiece   :: s -> [Text]
```

PathPiece-Instanzen für Int, Integer, String, Text und PathMultiPiece-Instanzen für [String] und [Text] vordefiniert

Fallstrick: read kann Ausnahme werfen! Meist reicht schon

```
maybeRead :: Read a => String -> Maybe a
maybeRead s | [(x,"")] <- reads s = Just x
            | otherwise           = Nothing
```

siehe auch readMay aus Classy-Prelude

Routing

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                  RootR     GET
  /blog/help         BlogHelpR GET
  /blog/#Int         BlogPostR GET POST
  /wiki/*WikiPfad    WikiR
  /static            StaticR   Static getStatic
|]
```

Pfade werden automatisch standardisiert, gereinigt und zerlegt.

Durch Überschreiben von Default-Funktionen der `Yesod`-Typklasse kann man dies aber auch anpassen:

- `joinPath` fügt Pfadteile wieder zusammen
- `cleanPath` kümmert sich um doppelte `/`, usw.

Überlappende Routen

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                RootR     GET
  !/blog/help      BlogHelpR GET
  !/blog/#Int      BlogPostR GET POST
  /wiki/*WikiPfad  WikiR
  /static          StaticR   Static getStatic
|]
```

- Yesod kann nicht inferieren, dass `/blog/help` und `/blog/#Int` nicht überlappen
- Überlappende Routen wie etwa `/blog/#Int` und `/blog/#Text` erzeugen eine Fehlermeldung
  Dies kann mit `!` am Anfang verhindert werden
- Von oben-nach-unten erst geparste Route wird eingeschlagen
  sonst 404 Page Not Found, konfigurierbar über Foundation

Datentyp für Route

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                  RootR     GET
  !/blog/help        BlogHelpR GET
  !/blog/#Int        BlogPostR GET POST
  /wiki/*WikiPfad    WikiR
  /static            StaticR   Static getStatic
|]
```

Der zweite Teil definiert den *Konstruktor* für die typsichere URL
Deklaration für Datentyp `Route App` wird erzeugt

erzeugter Code mit `-ddump-splices` ansehbar

- Kann direkt in URL Interpolation `@{ }` verwendet werden
  Argumente gemäß dynamischen Pfad, z.B. `@{BlogPostR 7}`
- Endung mit "R" für "Resource" ist keine zwingende Konvention

  Großbuchstabe am Anfang für Konstruktor aber schon

Datentyp für Route

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                   RootR     GET
  !/blog/help         BlogHelpR GET
  !/blog/#Int         BlogPostR GET POST
  /wiki/*WikiPfad     WikiR
  /static             StaticR   Static getStatic
|]
```

Der zweite Teil definiert den *Konstruktor* für die typsichere URL Deklaration für Datentyp `Route <FoundationType>` erzeugt

erzeugter Code mit `-ddump-splices` ansehbar

- Kann direkt in URL Interpolation `@{ }` verwendet werden
  Argumente gemäß dynamischen Pfad, z.B. `@{BlogPostR 7}`
- Endung mit "R" für "Resource" ist keine zwingende Konvention
  Großbuchstabe am Anfang für Konstruktor aber schon

Advanced Routing

Route-Attributes: Als letzte Parameter dürfen mehre mit `!` beginnende Text-Tags angegeben werden. Mit `routeAttrs` können diese Tags zu jedem Wert des Routen Typs abgefragt werden.

```
/admin    AdminR   GET      !admin
/lecture LectureR GET POST !admin !lecturer
```

Hierarchische Routen: Einrückung erlaubt hierarchische Routen Deklarationen, was zu Vereinfachung beim Handling gemeinsamer Routen führen kann:

```
/admin AdminR:
  /1 AOneR GET
  /2 ATwoR GET
```

Der Konstruktor `AdminR` bekommt ein weiteres Argument, welches hier `AOneR` oder `ATwoR` sein muss. Prüfung von Admin-Rechten könnte hier dann auf allen Routen erfolgen, welche auf `AdminR _` matchen.

Handling

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                 RootR     GET
  !/blog/help       BlogHelpR GET
  !/blog/#Int       BlogPostR GET POST
  /wiki/*WikiPfad   WikiR
  /static           StaticR   Static getStatic
|]
```

Dritter Teil spezifiziert entweder:

1. Erlaubte HTTP-Anfragen GET,PUT,POST,DELETE,...
2. Keine Angabe, d.h. alle HTTP-Anfragen erlaubt
   werden dann über einen gemeinsamen Handler abgewickelt
3. Foundation Typ einer Yesod-Subsite, welche die Anfrage beantwortet; und eine Funktion, welche Wert des aktuellen Foundation Typs in den Wert des Foundation Typs der Subsite umrechnen kann. Meistens nur eine Record-Projektion

Handling

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                 RootR     GET
  !/blog/help       BlogHelpR GET
  !/blog/#Int       BlogPostR GET POST
  /wiki/*WikiPfad   WikiR
  /static           StaticR   Static getStatic
|]
```

Für alle Anfragen muss ein **Handler** definiert werden, wobei der Name `<AnfrageTyp>` ++ `<Resource>` sein muss:

```
getRootR      :: Handler Html
getBlogHelpR  :: Handler Html
getBlogPostR  :: Int -> Handler Html
postBlogPostR :: Int -> Handler Html
handleWikiR   :: [WikiPfad] -> Handler Html
```

oder `handle` ++ `<Resource>`, falls alle HTTP-Anfragen erlaubt.

Subsite Handling

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                    RootR     GET
  !/blog/help          BlogHelpR GET
  !/blog/#Int          BlogPostR GET POST
  /wiki/*WikiPfad      WikiR
  /static              StaticR   Static getStatic
|]
```

- Route `/static` wird durch eine **Subsite** geleitet, welche darunterliegende Routen und Handler selbst definiert
- Im Beispiel ist `Static` der Foundation-Type der Subsite
- Angegebene Funktion `getStatic` muss Wert des Typs `Static` aus aktuellem Foundation-Type generieren können
- Scaffolding mit `yesod init` definiert Subsite für statische Resourcen um besseres Caching zu ermöglichen

Handler Monade

Handler-Funktionen leben in der Handler-Monade: `Handler Html`

```
type Handler a = HandlerFor App (IO a)
data HandlerFor site a = ...
```

`Handler` ist ein Typsynonym für einen Datentyp mit

`site`	Foundation Typ
`a`	Rückgabewert der Monade

Anstellte von `Html` kann ein Handler auch Werte anderer Typen liefern, zum Beispiel eine Grafikdatei, CSS, JSON, etc.
Der content type muss jedoch bekannt sein, d.h. Handler müssen als Ergebnistyp eine Instanz von `ToTypedContent` liefern

Handling nach Content

Es ist auch möglich, Unterschiedliche Repräsentation je nach MIME-typ über eine URL abzuwickeln:

```
mkYesod "App" [parseRoutes|
  /person PersonR GET
|]

getPersonR :: Handler TypedContent
getPersonR = selectRep $ do
  provideRep $ return [shamlet| <p>Name #{name}, Age #{age} |]
  provideRep $ return $ object [ "name" .= name, "age" .= age ]
 where
  name = "Steffen" :: Text
  age  = 40 :: Int
```

Für Anfrage `/person.html` wird HTML ausgeliefert und
für Anfrage `/person.json` hier das Gleiche in JSON. `Data.Aeson`

Handler Monade

Wichtige Funktionen der `Handler`-Monade:

`getYesod`	Wert des Foundation Typ, z.B. Parameter auslesen
`getUrlRender`	Renderer für Werte des `Route`-Typs
`getRequest`	Anfrage im Roh-Format
`liftIO`	zum Ausführen von IO-Aktionen
`sendFile`	Datei versenden
`setHeader`	Antwort-Header festlegen
`redirect`	Umleitung zu anderer Ressource

`notFound`, `permissionDenied` für explizite Fehlermeldung

`setCookie`, `lookupCookie` Cookies bearbeiten

`Handler` ist auch Instanz von `MonadLogger` für Logging.
Erzeugen von Log-Messages innerhalb von Templates mit
`$logError`, `$logWarn`, `$logInfo`, `$logDebug`

Webformulare

Webformulare erlauben dem Benutzer, Daten zu übermitteln.

Dies erfordert:

- Darstellung der Formulare in HTML
- Zuordnung der übermittelten Daten
- Konvertierung UTF8-Strings zu Haskell Datentypen
- Prüfungen, ob Daten im erlaubten Bereich sind
- JavaScript zu Daten-Prüfung und Bearbeitung auf dem Client
- Kombination verschiedener Formulare (-Teile)

Paket `yesod-form` stellt dies für uns bereit

Webformular Varianten in Yesod:

Applikativ Einfach zu Programmieren,
aber Kontrolle über Aussehen ist eingeschränkt

Monadisch Flexibel gestaltbare Formulare,
Verwendung jedoch etwas komplizierter

Input Spezialfall ohne HTML-Darstellung; hauptsächlich
zur Verwendung mit bestehenden Formularen

Es ist möglich, applikative Formulare automatisch in monadische umzuwandeln, und manchmal auch umgekehrt.

siehe `aformToForm` und `formToAForm`

Konvention in Yesod: Präfix je nach Variante `a`, `m` oder `i`

Z.B. Pflichtfeld in applikativen Formular erhält man mit `areq`,
ein optionales Feld in einem monadischen Formular mit `mopt`.

Applikative Formularfelder

```
data Car = Car { carModel :: Text, carYear  :: Int
                 carColor :: Maybe Text
               } deriving Show
carAForm :: AForm Handler Car
carAForm = Car
    <$> areq textField "Model" Nothing
    <*> areq intField  "Year"  (Just 1994)
    <*> aopt textField "Color" Nothing
```

Wir erklären, wie ein Wert des Typs `Car` erstellt wird:

- `areq` für erforderliche Felder,
  `aopt` für optionale Felder eines `Maybe`-Typen
- *1. Argument* definiert Typ durch Instanz der Klasse `Field` und erklärt damit dessen Parser; für viele Typen vordefiniert
- *2. Argument:* Bezeichner, Tooltip, id- und name-Attribut gegeben durch eine Instanz der Klasse `FieldSettings`
- *3. Argument:* Optionaler Default Wert gegeben als `Maybe`

Applikative Formularfelder

```
data Car = Car { carModel :: Text, carYear  :: Int
                 carColor :: Maybe Text
               } deriving Show
carAForm :: AForm Handler Car
carAForm = Car
    <$> areq textField "Model" Nothing
    <*> areq intField  "Year"  (Just 1994)
    <*> aopt textField "Color" Nothing
```

Wir erklären, wie ein Wert des Typs `Car` erstellt wird:

- `areq` für erforderliche Felder
  `ao`
- *1.*
  un

> Für `FieldSettings` ist eine Instanz zur Klasse `IsString` definiert, d.h. dank GHC-Erweiterung `OverloadedStrings` können solche Werte aus String-Konstanten generiert werden.

- *2. Argument:* Bezeichner, Tooltip, id- und name-Attribut gegeben durch eine Instanz der Klasse `FieldSettings`
- *3. Argument:* Optionaler Default Wert gegeben als `Maybe`

Applikatives Formular darstellen

```
carForm :: Html -> MForm Handler (FormResult Car, Widget)
carForm = renderTable carAForm

getCarR :: Handler Html
getCarR = do (widget, enctype) <- generateFormPost carForm
             defaultLayout [whamlet|
    <h2>Form Demo
    <form method=post action=@{CarR} enctype=#{enctype}>
      ^{widget}
      <button>Submit
  |]
```

- Umwandeln von `AForm` in `MForm` mit `renderTable`, `renderDivs`, oder `renderBootstrap` – entscheidet über Layout des Formulars
- Erzeugen des Formulars mit `generateFormGet` oder `generateFormPost`
- `form`-Tag und Knopf zum Absenden noch nicht enthalten, damit Formulare kombiniert werden können

Applikatives Formular auswerten

```
postCarR :: Handler Html
postCarR = do
  ((result,widget), enctype) <- runFormPost carForm
    case result of
      FormSuccess car -> defaultLayout [whamlet|
              <h2>Car received:
              <p>#{show car}  |]
      _ -> defaultLayout [whamlet|
          <h2>Fehler! Nochmal eingeben:
          <form method=post action=@{CarR} enctype=#{enctype}>
            ^{widget}
            <button>Abschicken
        |]
```

Ausführen des Formulars mit `runFormGet` oder `runFormPost`

weitere Varianten möglich, z.B. `runFormPostNoNonce`

Applikatives Formular auswerten

```
postCarR :: Handler Html
postCarR = do
  ((result,widget), enctype) <- runFormPost carForm
    case result of
      FormSuccess car -> defaultLayout [whamlet|
              <h2>Car received:
              <p>#{show car}  |]
      _ -> defaultLayout [whamlet|
          <h2>Fehler! Nochmal eingeben:
          <form method=post action=@{CarR} enctype=#{enctype}>
            ^{widget}
            <button>Abschicken
        |]
```

`result` hat 3 Möglichkeiten:

- `FormSuccess a` erfolgreicher Wert
- `FormFailure [Text]` Parsen fehlgeschlagen
- `FormMissing` keine Daten vorhanden

Applikatives Formular auswerten

```
postCarR :: Handler Html
postCarR = do
  ((result,widget), enctype) <- runFormPost carForm
    case result of
      FormSuccess car -> defaultLayout [whamlet|
              <h2>Car received:
              <p>#{show car}  |]
      _ -> defaultLayout [whamlet|
          <h2>Fehler! Nochmal eingeben:
          <form method=post action=@{CarR} enctype=#{enctype}>
            ^{widget}
            <button>Abschicken
        |]
```

Fehlermeldung nicht wie hier ignorieren, sondern dem Benutzer anzeigen!

`result` hat 3 Möglichkeiten:

- `FormSuccess a` erfolgreicher Wert
- `FormFailure [Text]` Parsen fehlgeschlagen
- `FormMissing` keine Daten vorhanden

Applikatives Formular auswerten

```
postCarR :: Handler Html
postCarR = do
 ((result,widget), enctype) <- runFormPost carForm
   case result of
    FormMissing -> addMessage "i" "No data" >> redirect getCarR
    FormFailure msgs -> defaultLayout [whamlet|
        <h2>Formular Fehler:
        <ul>
          $forall msg <- msgs
            <li>#{msg}
        <form method=post action=@{CarR} enctype=#{enctype}>
          ^{widget}
          <button>Abschicken      |]
```

Im Fehlerfall das Formular wiederholen, damit der Benutzer Fehler im ausgefüllten Formular korrigieren kann!

`result` hat 3 Möglichkeiten:

- `FormSuccess a` erfolgreicher Wert
- `FormFailure [Text]` Parsen fehlgesch
- `FormMissing` keine Daten vorhanden

Applikative Formularfelder

```
data Car = Car { carModel :: Text, carYear  :: Int
                 carColor :: Maybe Text
               } deriving Show
carAForm :: AForm Handler Car
carAForm = Car
    <$> areq textField "Model" Nothing
    <*> areq intField  "Year"  (Just 1994)
    <*> aopt textField "Color" Nothing
```

Wir erklären, wie ein Wert des Typs `Car` erstellt wird:

- `areq` für erforderliche Felder,
  `aopt` für optionale Felder eines `Maybe`-Typen
- *1. Argument* definiert Typ durch Instanz der Klasse `Field` und erklärt damit dessen Parser; für viele Typen vordefiniert
- *2. Argument:* Bezeichner, Tooltip, id- und name-Attribut gegeben durch eine Instanz der Klasse `FieldSettings`
- *3. Argument:* Optionaler Default Wert gegeben als `Maybe`

Applikative Formularfelder

```
data Car = Car { carModel :: Text, carYear  :: Int
                 carColor :: Maybe Text
               } deriving Show
carAForm :: AForm Handler Car
carAForm = Car
    <$> areq textField "Model" Nothing
    <*> areq intField  "Year"  (Just 1994)
    <*> aopt textField "Color" Nothing
```

Wir erklären, wie ein Wert des Typs `Car` erstellt wird:

- `areq` für erforderliche Felder
  `ao`
- *1.*
  un
- *2. Argument:* Bezeichner, Tooltip, id- und name-Attribut gegeben durch eine Instanz der Klasse `FieldSettings`
- *3. Argument:* Optionaler Default Wert gegeben als `Maybe`

Für `FieldSettings` ist eine Instanz zur Klasse `IsString` definiert, d.h. dank GHC-Erweiterung `OverloadedStrings` können solche Werte aus String-Konstanten generiert werden.

Defaults für Applikative Felder

```
data Car = Car { carModel :: Text, carYear  :: Int
                 carColor :: Maybe Text
               } deriving Show
carAForm :: Maybe Car -> AForm Handler Car
carAForm = Car
    <$> areq textField "Model" (carModel <$> mcar)
    <*> areq intField  "Year"  (carYear  <$> mcar)
    <*> aopt textField "Color" (carColor <$> mcar)
```

Es ist oft sinnvoll ein, alle Standardwerte als kompletten Wert des Datentyps zu übergeben

Aus diesem Grund wird in Kauf genommen, dass ansonsten optionale Standardvorgaben in ein doppeltes `Maybe` verpackt werden müssen:

```
  <*> aopt textField "Color" (Just $ Just "Black")
```

Spezialisierte Eingabeprüfung

```
data Car = Car { carModel :: Text, carYear  :: Int
                 carColor :: Maybe Text
               } deriving Show

carAForm :: Maybe Car -> AForm Handler Car
carAForm = Car
    <$> areq textField    "Model" (carModel <$> mcar)
    <*> areq carYearField "Year"  (carYear  <$> mcar)
    <*> aopt textField    "Color" (carColor <$> mcar)
  where
    errorMessage :: Text
    errorMessage = "Your car is too old, get a new one!"

    carYearField = check validateYear intField

    validateYear y
        | y < 1990  = Left errorMessage
        | otherwise = Right y
```

Spezialisierte Eingabeprüfung

Prüfung der Eingabe erfolgt durch modifizierte Feldtypen.
Modul `Yesod.Form.Functions` bietet dazu viele Hilfsfunktionen:

```
check     :: (a -> Either msg a)     -> Field m a -> Field m aSource
checkBool :: (a -> Bool) -> msg      -> Field m a -> Field m aSource
checkM    :: (a -> m (Either msg a)) -> Field m a -> Field m aSource
```

Mit `checkBool` hätten wie beispielsweise schreiben können:

```
carYearField = checkBool (>= 1990) errorMessage intField
```

Prüfungen unter Verwendung der IO-Monade sind ebenfalls möglich mit `checkM`, z.B. um das aktuelle Datum zu ermitteln und zu prüfen, ob das eingegebene Datum in der Zukunft liegt

Eingabeprüfung mit IO

```
carYearField = checkM inPast $ checkBool (>= 1990) errorMessage

inPast y = do
    thisYear <- liftIO getCurrentYear
    return $ if y <= thisYear
        then Right y
        else Left ("You have a time machine!" :: Text)

getCurrentYear :: IO Int
getCurrentYear = do
    now <- getCurrentTime
    let today = utctDay now
    let (year, _, _) = toGregorian today
    return $ fromInteger year
```

Fallstricke Eingabeprüfung

Aufgrund der generellen Unterstützung von Yesod für Internationalisierung (Abk. "i18n") werden zur erfolgreichen Kompilation oft zusätzliche Angaben gebraucht:

- `errorMessage :: Text` explizite Typangabe oft erforderlich, da die Fehlermeldung ja generell Sprachabhängig ist

- Ebenso ist folgende Instanz-Deklaration notwendig:

  ```
  instance RenderMessage App FormMessage where
      renderMessage _ _ = defaultFormMessage
  ```

  Diese wird vom Gerüst-Tool automatisch erstellt und kann dann angepasst werden, falls Internationalisierung gewünscht wird.

Auswahllisten

```
data Car = Car { carModel :: Text, carYear :: Int
               , carColor :: Maybe Color } deriving Show

data Color = Red | Blue | Gray | Black
    deriving (Show, Eq)

carAForm :: Maybe Car -> AForm FoundT FoundT Car
carAForm mcar = Car
    <$> areq textField    "Model" (carModel <$> mcar)
    <*> areq carYearField "Year"  (carYear <$> mcar)
    <*> aopt (selectFieldList colors) "Color" Nothing
  where
    colors :: [(Text, Color)]
    colors = [("Rot", Red), ("Blau", Blue),
              ("Grau", Gray), ("Schwarz", Black)]
```

Funktion `selectFieldList` nimmt eine Liste von (`Text`,Wert)-Paaren und kreiert eine Auswahlliste.

Auswahllisten

```
data Car = Car { carModel :: Text, carYear :: Int
                 carColor :: Maybe Color } deriving Show

data Color = Red | Blue | Gray | Black
    deriving (Show, Eq, Bounded, Enum)

carAForm :: Maybe Car -> AForm FoundT FoundT Car
carAForm mcar = Car
    <$> areq textField    "Model" (carModel <$> mcar)
    <*> areq carYearField "Year"  (carYear <$> mcar)
    <*> aopt (selectFieldList colors) "Color" Nothing
  where
    colors :: [(Text, Color)]
    colors = [(pack $ show x,x) | x <- [minBound..maxBound]]
```

Funktion `selectFieldList` nimmt eine Liste von (`Text`,Wert)-Paaren und kreiert eine Auswahlliste.

Auswahlknöpfe

```
data Car = Car { carModel :: Text, carYear :: Int
                 carColor :: Maybe Color } deriving Show

data Color = Red | Blue | Gray | Black
    deriving (Show, Eq, Bounded, Enum)

carAForm :: Maybe Car -> AForm FoundT FoundT Car
carAForm mcar = Car
    <$> areq textField    "Model" (carModel <$> mcar)
    <*> areq carYearField "Year"  (carYear <$> mcar)
    <*> aopt (radioFieldList colors) "Color" Nothing
  where
    colors :: [(Text, Color)]
    colors = [(pack $ show x,x) | x <- [minBound..maxBound]]
```

Funktion `radioFieldList` nimmt eine Liste von (`Text`,Wert)-Paaren; also genau wie `selectFieldList` auch!

JavaScript im Formular

```
import Yesod.Form.Jquery
import Data.Time (Day)

data Car = Car { carModel :: Text, carYear :: Int
                 carRegistration :: Day
  } deriving Show

instance YesodJquery FoundT
 -- Default: jQuery Libraries at Google CDN

carAForm :: Maybe Car -> AForm FoundT FoundT Car
carAForm mcar = Car
    <$> areq textField    "Model" (carModel <$> mcar)
    <*> areq carYearField "Year"  (carYear <$> mcar)
    <*> areq (jqueryDayField def
        { jdsChangeYear = True -- give a year dropdown
        , jdsYearRange = "2012:-20"
        }) "Zulassung" Nothing
```

Input Formulare ohne Layout

```
data Person = Person { personName :: Text, personAge  :: Int } deriving Show

getHomeR :: Handler Html
getHomeR = defaultLayout
    [whamlet|
        <form action=@{InputR}>
            <p>
                My name is
                <input type=text name=name>
                and I am
                <input type=text name=age>
                years old.
                <input type=submit value="Introduce myself">
    |]

getInputR :: Handler Html
getInputR = do
    person <- runInputGet $ Person
                <$> ireq textField "name"
                <*> ireq intField "age"
    defaultLayout [whamlet|<p>#{show person}|]
```

Übereinstimmung der Name-Tags muss gewährleistet werden!

Input Forms

Input Formulare

- Übereinstimmung der Name-Tags muss gewährleistet werden! Insbesondere bei dynamisch generierten Formularen problematisch.
- `ireq`/`iopt` haben nur noch zwei Argumente: Feld-Typ und Feld-Name
- Wenn die übermittelten Daten nicht passen erfolgt eine Umleitung auf eine “Invalid Arguments” Fehlerseite

Monadische Formulare

- erlauben ebenfalls eigenes Layout
- kümmern sich für uns jedoch um einzigartige Name-Tags, usw.
- `mreq`/`mopt` funktionieren wie `areq`/`aopt`, die Namen der Eingabefelder werden jedoch ignoriert (das Layout wird ja explizit angegeben)

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                      width: 3em;
                                      }
                             |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHomeR = do ((res, widget), enctype) <- runFormGet personForm
              defaultLayout [whamlet|
                  <p>Result: #{show res}
                  <form enctype=#{enctype}>
                      ^{widget} |]
```

Monadische Formulare

Alle Felder des Formulars werden mit monadischen Funktionen `mreq`/`mopt` beschrieben:

```
do
  (nameRes, nameView) <- mreq textField "not used" Nothing
  (ageRes , ageView)  <- mreq intField  "not used" Nothing
```

Für jedes Feld erhalten wir 2 Rückgabewerte:

1. Ergebnis des Feldes `FormResult a`, um später das Gesamtergebnis zu bauen:
   `let personRes = Person <$> nameRes <*> ageRes`
2. `FieldView` des Feldes zur Anzeige, zum Einbau in Widgets:
   `^{fvInput  ageView}` — Input-Feld
   `##{fvId    ageView}` — Id des Feldes

FieldView und FieldSettings

```
(nameRes, nameView) <- mreq textField "not used" Nothing
```

Wert des Typs `FieldView` ist Record mit den Feldern `fvLabel`, `fvTooltip`, `fvId`, `fvInput`, `fvErrors`, `fvRequired`.

Wird primär aus den Vorschlägen des zweiten Argument von `mreq`/`mopt` erzeugt, welches vom Typ `FieldSettings` sein muss.

Werte von `FieldSettings` können aus Strings erzeugt werden:

```
fromString "not used" == FieldSettings
  { fsLabel   = "not used" -- für Applikative Formulare
  , fsTooltip = Nothing
  , fsId      = Nothing
  , fsName    = Nothing
  , fsAttrs   = []
  }
```

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                        width: 3em;
                                        }
                               |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHomeR = do ((res, widget), enctype) <- runFormGet personForm
              defaultLayout [whamlet|
                  <p>Result: #{show res}
                  <form enctype=#{enctype}>
                      ^{widget} |]
```

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                      width: 3em;
                                      }
                             |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHome                                                          m
```

Felder werden durch zwei Teile beschrieben:

1. `FormResult`
2. `FormView`

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                      width: 3em;
                                      }
                             |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHome                                                        m
```

Felder werden durch zwei Teile beschrieben:

1. FormResult
2. FormView

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                          width: 3em;
                                          }
                                  |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHome                                                    m
```

Felder werden durch zwei Teile beschrieben:

1. FormResult
2. FormView

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                       width: 3em;
                                       }
                              |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHomeR = do ((res, widget), enctype) <- runFormGet personForm
              defaultLayout [whamlet|
                  <p>Result: #{show res}
                  <form enctype=#{enctype}>
                      ^{widget} |]
```

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                        width: 3em;
                                        }
                                   |]
                    [whamlet|
                      #{extra}
                       <p> Hello, my name is ^{fvInput nameView} and I am #
                           ^{fvInput ageView} years old. #
                           <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHomeR = do ((res, widget), enctype) <- runFormGet personForm
              defaultLayout [whamlet|
                  <p>Result: #{show res}
                  <form enctype=#{enctype}>
                      ^{widget} |]
```

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                        width: 3em;
                                        }
                               |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
```

Das extra Argument muss irgendwo ins Formular eingebaut werden.
Bei GET Formularen signalisiert es, dass das Formular abgesendet wurde.
Bei POST Formularen dient es zur Verhinderung von Cross-Site-Request-Forgery Angriffen.

```
                  <form enctype=#{enctype}>
                      ^{widget} |]
```

Sessions

HTTP kennt wie Haskell keinen Zustand   z.B. gut für Caching

Webapplikation benötigen aber Zustand: Login, ShoppingCart, etc.

Eine Lösung dazu sind **Sitzungen/Sessions**, d.h. eine kleine Menge an Daten (z.B. Sitzung-ID) welche der Browser *mit jeder Anfrage* an den Webserver übermittelt.

- Benutzerdaten mit Sitzungen zu speichern skaliert gut mit mehreren Servern, da jeder Request in sich abgeschlossen ist und keine zentrale Koordination/Datenbank benötigt wird
- Sitzungsdaten sollten möglichst klein sein   Datenbankschlüssel
- Statischer Content sollte von separater Domain kommen, um unnötige Übertragungen von Sitzungsdaten zu verhindern

  ⇒ `static` routing

Session Control

Yesod wendet per Default automatisch *Verschlüsselung* und *Signatur* von Sitzungsdaten an, um Manipulationen zu verhindern. Sitzung verfallen automatisch nach 2 Stunden. Dies wird alles in der Yesod-Instanz des Foundation Typs eingestellt:

```
instance Yesod App where
  makeSessionBackend _ = Just <$>
      defaultClientSessionBackend minutes file
    where minutes = 2 * 60
          file    = "client-session-key.aes"
  -- Sitzungen komplett deaktivieren:
  -- makeSessionBackend _ = return Nothing
```

- Schlüssel in separater Datei gespeichert   ggf. automatisch gen.
- Ablaufdatum der Sitzung wird mit jedem Request erneuert. Yesod ignoriert abgelaufene Sitzungen automatisch.
- IP-Adresse einer Sitzung wird überprüft   ggf. abschaltbar

Session Operationen

Sitzung ist eine *ungetypte* Map:

```
type SessionMap = Map Text ByteString
```

```
getSession       :: MonadHandler m => m SessionMap
```

Liefert gesamte Sitzungs-Map – inkl. Yesod-internes

```
lookupSessionBS :: MonadHandler m => Text -> m (Maybe
                                                   ByteString)
```

Schlägt Schlüssel nach

```
lookupSession   :: MonadHandler m => Text -> m (Maybe Text)
```

Schlägt Schlüssel nach

```
setSession      :: MonadHandler m => Text -> Text -> m ()
```

Setzt ein Schlüssel-Wert Paar

```
deleteSession   :: MonadHandler m => Text -> m ()
```

Löscht einen Schlüssel

```
clearSession    :: MonadHandler m => m ()
```

Löscht die gesamte Sitzungs-Map

Beispiel: Sessions

```
getHomeR :: Handler Html
getHomeR = do
    sess <- getSession
    defaultLayout [whamlet|
                    <form method=post>
                        <input type=text name=key>
                        <input type=text name=val>
                        <input type=submit>
                    <h1>#{show sess}
                |]

postHomeR :: Handler ()
postHomeR = do
    (key, mval) <- runInputPost $ (,) <$> ireq textField "key"
                                      <*> iopt textField "val"
    case mval of Nothing  -> deleteSession key
                 Just val -> setSession key val
    liftIO $ print (key, mval) --debug to konsole
    redirect HomeR
```

Messages

Post/Redirect/Get-Problem: Z.B. nach erfolgreichem Ausfüllen eines Formulars den Benutzer umleiten und gleichzeitig informieren, dass die Daten korrekt empfangen wurden.

Lösung jede Seite prüft Existenz eines speziellen Sitzungs-Feldes für Nachrichten. Yesod bietet eigene Schnittstelle dafür:

```
addMessage  :: MonadHandler m => Text -> Html -> m ()
```

Setzt eine Message in der Sitzung.

```
getMessages :: MonadHandler m => m [(Text, Html)]
```

Liest Messages aus und löscht diese, sofern vorhanden.

`defaultLayout` nutzt `getMessages` um ggf. Botschaft anzuzeigen
⇒ bei Überschreiben von `defaultLayout` die Behandlung von `getMessages` nicht vergessen!

Beispiel: Messages

```
getA = do
  page <- defaultLayout $ do
    [whamlet|
      You are at A
    |]
    links
  addMessage "info" "Previous: A"
  return page

getB = do
  msgs <- getMessages
  page <- defaultLayout $ do
    [whamlet|
      <p>You are at B
      $forall (cls,msg) <- msgs
        <p class="#{cls}">Message: #{msg}
    |]
    links
  addMessage "warning" "Previous: B"
  return page
```

Ultimate Destination

Erlaubt temporäre Umleitung eines Benutzers (z.B. für Login); danach wird Benutzer wieder zur ursprünglichen Seite geschickt

```
setUltDest ::(MonadHandler m, RedirectUrl (HandlerSite m) url) =>
            url -> m ()
```
Setzt Ultimate Destination

```
setUltDestCurrent :: MonadHandler m => m ()
```
Setzt Ultimate Destination auf aktuelle Seite, falls $\neq$ 404

```
setUltDestReferer :: MonadHandler m => m ()
```
Setzt Ultimate Destination auf Referer = vorherige Seite

```
redirectUltDest   :: ... =>  url -> m a
```
Führt redirect auf Ultimate Destination aus und löscht diese, falls gesetzt, sonst redirect auf angegebene URL

```
clearUltDest      :: MonadHandler m => m ()
```
Löscht Ultimate Destination.

Beispiel: Ultimate Destination

```
getSayHelloR = do -- Display name or request it
    mname <- lookupSession "name"
    case mname of
        Nothing -> do
            setUltDestCurrent
            addMessage "info" "Please tell me your name"
            redirect SetNameR
        Just name -> defaultLayout [whamlet|<p>Welcome #{name}|]

getSetNameR = defaultLayout -- Display form
    [whamlet|
        <form method=post>
            My name is #
            <input type=text name=name>
            . #
            <input type=submit value="Set name">
    |]

postSetNameR = do -- Evaluate Form & set in session
    name <- runInputPost $ ireq textField "name"
    setSession "name" name
```

Persistenz

Manchmal sollen Daten länger als eine Sitzung halten.

`Database.Persist` und `Yesod.Persistent` stellen dazu eine *typsichere* Schnittstelle für Standard-Datenbanken bereit. Typsicherheit gilt auch dann, wenn die eigentliche Datenbank selbst ungetypt ist!

- `Persistent` unterstützt verschiedene Datenbanken: SQLite, PostgreSQL, MySQL, MongoDB, ...
- `Persistent` führt viele SQL Migrationen automatisch aus

Modul `Database.Persist` ist *unabhängig von Yesod*, und kann generell zur Anbindung einer Datenbank an ein Haskell Programm verwendet werden.

Typ-Sichere Persistenz

Datenbanken werden mit TemplateHaskell spezifiziert:

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
        name String
        age Int
        deriving Show
    BlogPost
        title String
        authorId PersonId
    |]
```

Definiert Hilfsfunktionen und Haskell-Datentypen:

```
data Person { personName :: String, personAge :: Int }
        deriving (Show, Read, Eq)
type PersonId = Key Person

data BlogPost { blogPostTitle    :: String,
                blogPostAuthorId :: PersonId }
        deriving (Read, Eq)
```

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persistLowerCase|
    Person
        name String
        age Int Maybe
        deriving Show
    BlogPost
        title String
        authorId PersonId
        deriving Show
    |]

main :: IO ()
main = runSqlite "dbfile.sql" $ do
    runMigration migrateAll
    johnId <- insert $ Person "John Doe" $ Just 35
    janeId <- insert $ Person "Jane Doe" Nothing

    insert $ BlogPost "My fr1st p0st" johnId
    insert $ BlogPost "One more for good measure" johnId

    oneJohnPost <- selectList [BlogPostAuthorId ==. johnId] [LimitTo 1]
    liftIO $ print (oneJohnPost :: [Entity BlogPost])
    john <- get johnId
    liftIO $ print (john :: Maybe Person)
    delete janeId
    deleteWhere [BlogPostAuthorId ==. johnId]
```

Benutzerspezifische Datenbankfelder

Feldtypen müssen Instanzen der Klasse `PersistField` sein.
Für Enumerations kann die Instanz-Deklaration mit einer TemplateHaskell Funktion automatisch abgeleitet werden:

```
data Employment = Employed | Unemployed | Retired
    deriving (Show, Read, Eq)
derivePersistField "Employment"

-- Andere Datei:
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
[persistLowerCase|
Person
    name String
    employment Employment
|]
```

Konstruktoren werden in der Datenbank als String gespeichert, welche mit `Show` und `Read` verarbeitet werden. Dies erlaubt nachträgliche Erweiterung der Konstruktoren.

Benutzerspezifische Datenbankfelder

Feldtypen müssen Instanzen der Klasse `PersistField` sein.
Für Enumerations kann die Instanz-Deklaration mit einer TemplateHaskell Funktion automatisch abgeleitet werden:

```
data Employment = Employed | Unemployed | Retired
    deriving (Show, Read, Eq)
derivePersistField "Employment"

-- Andere Datei:
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
[persistLowerCase|
Person
    name String
    employment Employment
|]
```

Alternative
Speicherung innerhalb der Datenbank im JSON Format
`derivePersistFieldJSON`

Konstruktoren werden in der Datenbank als String gespeichert, welche mit `Show` und `Read` verarbeitet werden. Dies erlaubt nachträgliche Erweiterung der Konstruktoren.

Uniqueness

Zeilen, welche mit Großbuchstaben beginnen, spezifizieren Einschränkung zu Einzigartigkeit von Datenbankeinträgen:

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
      firstName String
      lastName String
      age Int
      UniquePerson firstName
      deriving Show
  |]
```

Es wird ein Konstruktor generiert, der gezieltes Nachschlagen mit der Funktion `getBy` erlaubt, das Feld wird also auch zum Schlüssel:

```
getBy $ UniquePerson "Steffen"
```

Uniqueness

Zeilen, welche mit Großbuchstaben beginnen, spezifizieren Einschränkung zu Einzigartigkeit von Datenbankeinträgen:

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
      firstName String
      lastName String
      age Int
      UniquePerson firstName
      deriving Show
  |]
```

Der eigentliche Schlüssel bleibt unverändert. Mit `Primary firstName` würde das Feld zum echten Datenbankschlüssel werden.

Es wird ein Konstruktor generiert, der gezieltes Nachschlagen mit der Funktion `getBy` erlaubt, das Feld wird also auch zum Schlüssel:

```
getBy $ UniquePerson "Steffen"
```

UNIQUENESS

Zeilen, welche mit Großbuchstaben beginnen, spezifizieren Einschränkung zu Einzigartigkeit von Datenbankeinträgen:

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
      firstName String
      lastName String
      age Int
      UniquePerson firstName lastName
      deriving Show
  |]
```

Es wird ein Konstruktor generiert, der gezieltes Nachschlagen mit der Funktion `getBy` erlaubt, das Feld wird also auch zum Schlüssel:

```
getBy $ UniquePerson "Steffen" "Jost"
```

Werden mehrere Felder angegeben, dann muss nur die entsprechende Kombination einzigartig sein.

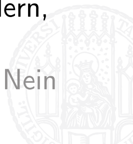
Optionale Felder

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
      firstName String
      lastName String
      age Int Maybe
      deriving Show
  |]
```

`Maybe` wird ganz *hinten* angestellt und macht das Feld optional in der Datenbank "nullable"

Achtung: Uniqueness funktioniert nur mit nicht-optionalen Feldern, da SQL nicht festlegt ob `NULL==NULL` gilt

Haskell: Ja, PostgreSQL: Nein



Nachträgliche Erweiterung

`Persist` kann sich auch um nachträgliche Änderungen an der Datenbank kümmern und übernimmt die **Migration**

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
      firstName String
      lastName String
      age Int Maybe
      timestamp UTCTime default=CURRENT_TIME
      deriving Show
|]
```

- Hinzufügen optionaler Felder ist problemlos   auf NULL gesetzt
- Mit `default` können Standardwerte vorgegeben werden
  Achtung: Dies ist Datenbank spezifisch, z.B. `CURRENT_TIME` ist hier eine spezielle Funktion der Datenbank

Datenbank Migration

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
        name String
        age Int
  |]
```

Automatische Migration mit Funktion `runMigration` bei:

- Zusätzliche Datentypen hinzufügen
- Zusätzliche Felder mit Default hinzufügen
- Typwechsel bei Felder, sofern Konversion möglich
  z.B. Wechsel zwischen Optional- und Pflichtfeld

Manuelle Intervention notwendig bei: `runMigrationUnsafe`

- Felder löschen
- Umbenennung von Felder oder Datentypen

Aktionen werden auf `stderr` protokolliert;
Migrations-Vorschau mit `printMigration` möglich

Datenbank Schnittstelle

```
main = runSqlite ":memory:" $  do
  runMigration $ migrateAll -- ggf. Migration durchführen
  steffenId <- insert $ Person "Steffen" 37 -- Einfügen
  steffen   <- get steffenId                -- Abfragen
  liftIO $ print steffen
```

`runSqlite` Datenbank einbinden, je nach Argument:

- `":memory:"` Datenbank im Speicher
- `"myfile.sql"` Datenbank in Datei

Genau eine Datenbank Transaktion pro Aufruf von `runSqlite`
Atomar, d.h. alle Aktionen bei Ausnahme zurückgenommen

`runMigration` Default-Migration; mindestens einmal bei Erstellen einer neuen Datenbank durchführen!

Datenbank Schnittstelle

Die Klasse `PersistStore b m` definiert u.a.:

`insert  :: ... => val -> m (Key val)`
Wert in Datenbank einfügen

`get     :: ... => Key b val -> m (Maybe val)`
Schlüssel in Datenbank nachschlagen

`getBy   :: ... => Unique val -> m (Maybe (Entity val))`
Unique nachschlagen, liefert `Entity valId val`

`delete  :: ... => Key val -> m ()`
Schlüssel in Datenbank löschen

`repsert :: ... => Key val -> val -> m ()`
Schlüssel ggf. ersetzen

```
main = runSqlite ":memory:" $  do
  runMigration $ migrateAll
  steffenId <- insert $ Person "Steffen" 37
  steffen <- get steffenId
  liftIO $ print steffen
```

Datenbank Abfragen

Neben `get` und `getBy` gibt es noch echte Datenbank Abfragen:

```
selectList :: ... =>
    [Filter val] -> [SelectOpt val] -> m [Entity val]
```

Zwei Argumente:

- Liste von Filtern
- List von Auswahl Optionen

Beispiel: Alle Menschen zwischen 26 und 30 auswählen:

```
people25bis30 <- selectList
  [PersonAge >. 25, PersonAge <=. 30] []
```

Datenbank Abfragen

- Liste der Filter ist UND-Verknüpft
- Operatoren wie gewohnt, nur mit Punkt am Ende
- `!=.` anstelle von `/=.` wegen Namenskonflikt
- `<-.` und `/<-.` stehen für "element" und "nicht element"

Beispiel: Alle Menschen zwischen 26 und 30, oder deren Namen nicht "Adam" oder "Bonny" lautet, oder deren Alter genau 50 oder 60 beträgt:

```
people <- selectList
  (       [PersonAge >. 25, PersonAge <=. 30]
      ||. [PersonFirstName /<-. ["Adam", "Bonny"]]
      ||. ([PersonAge ==. 50] ||. [PersonAge ==. 60])
  )
  []
```

Datenbank Abfragen

Auswahl Optionen:

- `Asc` *`Feld`* für aufsteigend sortierte Ergebnisse
- `Desc` *`Feld`* für absteigenden sortierte Ergebnisse
- `LimitTo n` um Anzahl Ergebnisse zu Begrenzen
- `OffSetBy n` um die ersten n-Ergebnisse zu überspringen

```
let resultsPerPage = 10
selectList
    [ PersonAge >=. 18 ]
    [ Desc PersonAge
    , Asc PersonLastName
    , Asc PersonFirstName
    , LimitTo resultsPerPage
    , OffsetBy $ (pageNumber - 1) * resultsPerPage
    ]
```

Datenbank Abfragen

Alternative Abfragen:

```
selectList :: ... =>
    [Filter val] -> [SelectOpt val] -> m [Entity val]
```

liefert Ergebnis-Liste

```
selectFirst :: ... =>
    [Filter val] -> [SelectOpt val] ->  m (Maybe (Entity val))
```

liefert nur das erste Ergebnis

```
selectKeys :: PersistEntity val =>
    [Filter val] -> Source (ResourceT (b m)) (Key val)
```

liefert nur die Schlüssel der Ergebnisse

Direkte, rohe, typ-unsichere SQL Operationen sind auch möglich, z.B. für Joins

Datenbank Manipulationen

```
update :: PersistEntity val =>
  Key val -> [Update val] -> m ()
```

Datenbankwert verändern

```
updateWhere :: PersistEntity val =>
  [Filter val] -> [Update val] -> m ()
```

nur spezielle Werte verändern

```
deleteWhere :: PersistEntity val =>
  [Filter val] -> m ()
```

nur spezielle Werte löschen

```
personId <- insert $ Person "Steffen" "Jost" 39
update personId [PersonAge =. 40]

updateWhere [PersonFirstName ==. "Steffen"]
            [PersonAge +=. 1]
```

Operatoren: `=.`, `+=.`, `-=.`, `*=.`, `/=.`

Datenbanken Integration in Yesod

Datenbank/Yesod-Schnittstelle in `Yesod.Persist` definiert:

```
runDB :: YesodDB site a -> HandlerFor site a
```

Datenbank Zugriff in Handler Monade

```
get404 :: ... => Key val -> m val
```

Wie `get`, nur liefert direkt 404 bei fehlschlag `getBy404`

- `YesodPersist`-Instanz des Foundation Types hält fest, welche Datenbank verwendet wird.
- Foundation Typ erhält ein Argument für die Datenbank, damit diese überall zugänglich ist.
- Yesod Scaffolding Tool kümmert sich bereits um alles

Beispiel: Datenbank in Yesod

Minimales Yesod-Beispiel ändern/erweitern um:

```
data App = App ConnectionPool -- Parameter für Foundation

instance YesodPersist PersistTest where
    type YesodPersistBackend PersistTest = SqlBackend
    runDB action = do
        App pool <- getYesod
        runSqlPool action pool

openConnectionCount :: Int
openConnectionCount = 10

main :: IO ()
main = runStderrLoggingT $ withSqlitePool "myfile.db3"
  openConnectionCount $ \pool -> liftIO $ do
    runResourceT $ flip runSqlPool pool $ do
        runMigration migrateAll
        insert $ Person "Michael" "Snoyman" 26
    warp 3000 $ PersistTest pool

getPersonR :: PersonId -> Handler String
getPersonR personId = do  person <- runDB $ get404 personId
                          return $ show person
```

Datenbank Zugriff in Widgets

Keine Datenbankanfragen innerhalb Widgets erlaubt:

```
[whamlet|
  <ul>
    $forall Entity blogid blog <- blogs
      $with author <- runDB $ get404 $ blogAuthor -- Error
        <li>
          <a href=@{BlogR blogid}>
            #{blogTitle blog} by #{authorName author}
|]
```

PROBLEM:

Innerhalb des Widgets befinden wir uns nicht mehr in der `Handler`-Monade!

Datenbank Zugriff in Widgets

Lösung Abfrage vorher durchführen:

```
getHomeR :: Handler Html
getHomeR = do blogs <- runDB $ selectList [] []
              defaultLayout $ do
                setTitle "Blog posts"
                [whamlet|
                  <ul>
                    $forall blogEntity <- blogs
                      ^{showBlogLink blogEntity}
                |]
showBlogLink :: Entity Blog -> Widget
showBlogLink (Entity blogid blog) = do
  author <- handlerToWidget $ runDB $ get404 $ blogAuthor blog
  [whamlet|
      <li>
          <a href=@{BlogR blogid}>
              #{blogTitle blog} by #{authorName author}
  |]
```

Monaden zusammenfassen!

Mehrere aufeinanderfolgende Datenbankzugriffe sollten möglichst zusammengefasst werden, um die Zugriffe zu beschleunigen:

```
getHomeR :: Handler Html
getHomeR do
   (p1Id,p2Id,list) <- runDB $ do
     pers1Id <- insert ...
     pers2Id <- insert ...
     list    <- selectList ...
     return (pers1Id, pers2Id, list)
```

anstatt

```
getHomeR do
   pers1Id <- runDB $ insert ...
   pers2Id <- runDB $ insert ...
   list    <- runDB $ selectList ...
```

Gilt allgemein:

nicht 5x hintereinander `liftIO`, sondern einmal `liftIO $ do` ...

Weitere Themen:

Yesod

- Authentifizierung: Wer macht den Zugriff?
- Autorisierung: Wer darf welchen Zugriff machen?
- Internationalisierung:
  Eine Seite in mehreren Sprachen ausliefern
- Subsites:
  Webapp aus Bausteinen zusammensetzen

Persist

- Esqueleto: Separate Zusatz-Bibliothek
  Typsichere EDSL generiert rohe SQL-Anfragen
  z.B. nützlich für Joins

Esqueleto

Das Paket `esqueleto` bietet eine eingebettete Domänen-spezifische Sprach (EDSL) zur Formulierung von SQL Anfragen.

Für die Fragmente der SQL Sprache für Datenbanken gibt es Haskell Ausdrücke. Das Typsystem von Haskell kann uns somit helfen, Fehler in unseren SQL Ausdrücken zu finden.
Der Haskell-Code für SQL soll möglichst wie echtes SQL aussehen.

Esqueleto unterstützt verschiedene Varianten, z.B. PostgreSQL oder SQLite. Die formulierten Anfragen sind spezifisch für jede Variante.

Esqueleto erlaubt die Konstruktion komplexer SQL Anfragen, inklusive Joins. Rohes SQL ist möglich, hebelt aber die Typsicherheit aus.

Esqueleto – Beispiel

Liste mit Namen offener Arbeitsblätter und zugehörigem Kursnamen.
Kurse und Blätter sind getrennte Tabellen in der Datenbank, weshalb wir einen SQL-Join benötigen:

```
[persist| Course
            name      String
          Sheet
            name      String
            course    CourseId
            activeTo  UTCTime   |]
```

```
import qualified Database.Esqueleto as E
openSheets :: Handler Html
openSheets = do
  now <- liftIO getCurrentTime
  courseSheets <- runDB $ E.select $
    E.from $ \(course `E.InnerJoin` sheet) -> do
      E.on $ course E.^. CourseId E.==. sheet E.^. SheetCourse
      E.where_ $ sheet E.^. SheetActiveTo E.>=. E.val now
      return (course, sheet)
  defaultLayout $ [whamlet|
    <table>
      $forall (Entity cid course, Sheet sid sheet) <- courseSheets
        <tr><td>#{courseName course}</td><td>#{sheetName  sheet}</td>
   |]
```

Esqueleto – Beispiel

Liste mit Namen offener Arbeitsblätter und zugehörigem Kursnamen.
Kurse und Blätter sind getrennte Tabellen in der Datenbank, weshalb wir einen SQL-Join benötigen:

```
[persist| Course
            name        String
          Sheet
            name        String
            course      CourseId
            activeTo    UTCTime    |]
```

```
import Database.Esqueleto
openSheets :: Handler Html
openSheets = do
  now <- liftIO getCurrentTime
  courseSheets <- runDB $ select $
    from $ \(course `InnerJoin` sheet) -> do
      on $ course ^. CourseId ==. sheet ^. SheetCourse
      where_ $ sheet ^. SheetActiveTo >=. val now
      return (course, sheet)
  defaultLayout $ [whamlet|
    <table>
      $forall (Entity cid course, Sheet sid sheet) <- courseSheets
        <tr><td>#{courseName course}</td><td>#{sheetName  sheet}</td>
   |]
```

Quellenangabe

Dieses Kapitel zeigte Ideen und Code-Beispiele unter anderem aus folgenden Quellen:

- Michael Snoyman. “Haskell and Yesod”. O’Reilly, April 2012, ISBN 978-1-449-31697-6
- Dokumentation des Frameworks auf http://www.yesodweb.com
- Dokumentation von GHC auf https://www.haskell.org