Fortgeschrittene Funktionale Programmierung in Haskell

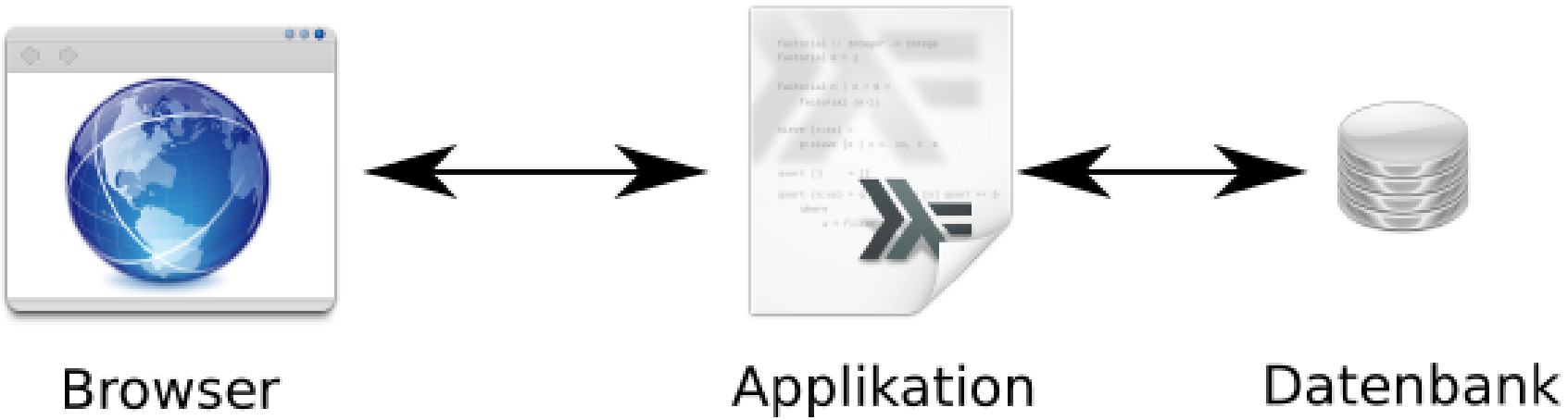
Jonas Betzendahl
Stefan Dresselhaus

Vorlesung 11: *Web-Development mit Yesod*
Stand: 24. Juni 2016

Was ist Web-Development?

Was ist Web-Development?

Im folgenden stellen wir die einzelnen Schritte vor, die beim Aufruf einer Website passieren. Wir werden uns hierbei von einer groben zu einer feinen Architektur entlanghangeln.



Das nutzen einer Webapplikation ist sehr simpel: Der User gibt eine URL an und erhält die zugehörigen Informationen.

Das nutzen einer Webapplikation ist sehr simpel: Der User gibt eine URL an und erhält die zugehörigen Informationen.

Daher macht es Sinn sich den Aufbau einer solchen URL genauer anzusehen:

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

Das nutzen einer Webapplikation ist sehr simpel: Der User gibt eine URL an und erhält die zugehörigen Informationen.

Daher macht es Sinn sich den Aufbau einer solchen URL genauer anzusehen:

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

Das Protokoll über das Kommuniziert wird. Im Webkontext meist `http` und https

Das nutzen einer Webapplikation ist sehr simpel: Der User gibt eine URL an und erhält die zugehörigen Informationen.

Daher macht es Sinn sich den Aufbau einer solchen URL genauer anzusehen:

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

Eine Subdomain, die (meist) einen anderen Server anspricht, als die eigentliche Domain.
Z.B. in `techfak.uni-bielefeld.de`

Das nutzen einer Webapplikation ist sehr simpel: Der User gibt eine URL an und erhält die zugehörigen Informationen.

Daher macht es Sinn sich den Aufbau einer solchen URL genauer anzusehen:

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

Die eigentliche Domain, die den Server identifiziert.

Das nutzen einer Webapplikation ist sehr simpel: Der User gibt eine URL an und erhält die zugehörigen Informationen.

Daher macht es Sinn sich den Aufbau einer solchen URL genauer anzusehen:

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

Die Top-Level-Domain, die zumeist rechtliche Konsequenzen hat.

Das nutzen einer Webapplikation ist sehr simpel: Der User gibt eine URL an und erhält die zugehörigen Informationen.

Daher macht es Sinn sich den Aufbau einer solchen URL genauer anzusehen:

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

Der Pfad zu der Ressource, die wir auf dem Server abrufen.

Das nutzen einer Webapplikation ist sehr simpel: Der User gibt eine URL an und erhält die zugehörigen Informationen.

Daher macht es Sinn sich den Aufbau einer solchen URL genauer anzusehen:

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

Der Name der Ressource, die wir abrufen.

Vorsicht:

`mydomain.tld/home` und `mydomain.tld/home/` sind vollkommen verschiedene Anfragen. Wird keine Ressource angegeben, so nimmt der Server die `default`-Route.

Das nutzen einer Webapplikation ist sehr simpel: Der User gibt eine URL an und erhält die zugehörigen Informationen.

Daher macht es Sinn sich den Aufbau einer solchen URL genauer anzusehen:

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

Parameter mit werten, die der Ressource beim Aufruf übergeben werden.
`?par1=val1&par2=val2&par3=val3` benutzt man bei mehreren Parametern.

Das nutzen einer Webapplikation ist sehr simpel: Der User gibt eine URL an und erhält die zugehörigen Informationen.

Daher macht es Sinn sich den Aufbau einer solchen URL genauer anzusehen:

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

Eine Anweisung an den Browser auf welchem Teil der Seite sich der User befindet. Diese wird nicht an den Server gesendet.

Das nutzen einer Webapplikation ist sehr simpel: Der User gibt eine URL an und erhält die zugehörigen Informationen.

Daher macht es Sinn sich den Aufbau einer solchen URL genauer anzusehen:

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

Dies ist der einzige Teil, der von unserem Server verarbeitet werden muss.

Unser Server braucht irgendeine Methode um Daten sinnvoll speicher zu können. Wir wollen nicht, dass alle User gelöscht sind, nur, weil der Strom ausgefallen ist.

Unser Server braucht irgendeine Methode um Daten sinnvoll speicher zu können. Wir wollen nicht, dass alle User gelöscht sind, nur, weil der Strom ausgefallen ist.
Eine Datenbank ist nichts weiter, als ein `State`, der im Hintergrund gehalten wird. Und immer, wenn wir in Haskell einen State haben, stecken wir ihn in eine Monade.

Unser Server braucht irgendeine Methode um Daten sinnvoll speicher zu können. Wir wollen nicht, dass alle User gelöscht sind, nur, weil der Strom ausgefallen ist.
Eine Datenbank ist nichts weiter, als ein `State`, der im Hintergrund gehalten wird. Und immer, wenn wir in Haskell einen State haben, stecken wir ihn in eine Monade.
Dies hat die üblichen Vorteile:

Unser Server braucht irgendeine Methode um Daten sinnvoll speicher zu können. Wir wollen nicht, dass alle User gelöscht sind, nur, weil der Strom ausgefallen ist.
Eine Datenbank ist nichts weiter, als ein `State`, der im Hintergrund gehalten wird. Und immer, wenn wir in Haskell einen State haben, stecken wir ihn in eine Monade.
Dies hat die üblichen Vorteile:

- ein fest definiertes Interface

Unser Server braucht irgendeine Methode um Daten sinnvoll speicher zu können. Wir wollen nicht, dass alle User gelöscht sind, nur, weil der Strom ausgefallen ist.
Eine Datenbank ist nichts weiter, als ein `State`, der im Hintergrund gehalten wird. Und immer, wenn wir in Haskell einen State haben, stecken wir ihn in eine Monade.
Dies hat die üblichen Vorteile:

- ein fest definiertes Interface
- Die Monade kümmert sich um alle angelegenheiten der Synchronisation

Unser Server braucht irgendeine Methode um Daten sinnvoll speicher zu können. Wir wollen nicht, dass alle User gelöscht sind, nur, weil der Strom ausgefallen ist.
Eine Datenbank ist nichts weiter, als ein `State`, der im Hintergrund gehalten wird. Und immer, wenn wir in Haskell einen State haben, stecken wir ihn in eine Monade.
Dies hat die üblichen Vorteile:

- ein fest definiertes Interface
- Die Monade kümmert sich um alle angelegenheiten der Synchronisation
- Das Typsystem bewahrt uns davor undefinierte Aktionen zu machen

Damit sind die Aufgaben des eigentlichen Servers klar:

- Interpretieren der Teil-URL

Damit sind die Aufgaben des eigentlichen Servers klar:

- Interpretieren der Teil-URL
- Aufrufen der richtigen Funktion

Damit sind die Aufgaben des eigentlichen Servers klar:

- Interpretieren der Teil-URL
- Aufrufen der richtigen Funktion
- Unter zuhilfenahme der Datenbank die Aktion ausführen

Damit sind die Aufgaben des eigentlichen Servers klar:

- Interpretieren der Teil-URL
- Aufrufen der richtigen Funktion
- Unter zuhilfenahme der Datenbank die Aktion ausführen
  Z.B. Daten auslesen oder Änderungen speichern

Damit sind die Aufgaben des eigentlichen Servers klar:

- Interpretieren der Teil-URL
- Aufrufen der richtigen Funktion
- Unter zuhilfenahme der Datenbank die Aktion ausführen
  Z.B. Daten auslesen oder Änderungen speichern
- Erstellen einer Antwort und zurücksenden dieser

Damit sind die Aufgaben des eigentlichen Servers klar:

- Interpretieren der Teil-URL
- Aufrufen der richtigen Funktion
- Unter zuhilfenahme der Datenbank die Aktion ausführen
  Z.B. Daten auslesen oder Änderungen speichern
- Erstellen einer Antwort und zurücksenden dieser

Also fangen wir an.

Verwaltung der Datenbank

Die Datenbank stellt unseren zentralen Speicher dar, allerdings wird und fast alles an Arbeit abgenommen.

Verwaltung der Datenbank

Die Datenbank stellt unseren zentralen Speicher dar, allerdings wird und fast alles an Arbeit abgenommen.

Wir müssen lediglich eine Record-Ähnliche Datendefinition liefern:

```
User
    ident Text
    name Text
    password Text Maybe --password ist nicht zwingend gesetzt
    lastLogin UTCTime
    UniqueUser ident    --ident ist eindeutig
    deriving Typeable   --deriving geht auch

Post
    content Text
    user UserId
    time UTCTime
```

Verwaltung der Datenbank

Die Datenbank stellt unseren zentralen Speicher dar, allerdings wird und fast alles an Arbeit abgenommen.

```
User
    ident Text
    name Text
    password Text Maybe --password ist nicht zwingend gesetzt
    lastLogin UTCTime
    UniqueUser ident    --ident ist eindeutig
    deriving Typeable   --deriving geht auch

Post
    content Text
    user UserId
    time UTCTime
```

Zunächst gibt es die Typen des jeweiligen Records analog zu

`data neuertyp = ...`

Verwaltung der Datenbank

Die Datenbank stellt unseren zentralen Speicher dar, allerdings wird und fast alles an Arbeit abgenommen.

```
User
    ident Text
    name Text
    password Text Maybe --password ist nicht zwingend gesetzt
    lastLogin UTCTime
    UniqueUser ident    --ident ist eindeutig
    deriving Typeable   --deriving geht auch

Post
    content Text
    user UserId
    time UTCTime
```

und die jeweigen benannten Felder

Verwaltung der Datenbank

Die Datenbank stellt unseren zentralen Speicher dar, allerdings wird und fast alles an Arbeit abgenommen.

```
User
    ident Text
    name Text
    password Text Maybe --password ist nicht zwingend gesetzt
    lastLogin UTCTime
    UniqueUser ident    --ident ist eindeutig
    deriving Typeable   --deriving geht auch

Post
    content Text
    user UserId
    time UTCTime
```

mit den entsprechenden Typen.

Diese Typen müssen aber Datenbank-Kompatibel sein - also in der Typklasse `PersistField` liegen.

Verwaltung der Datenbank

Die Datenbank stellt unseren zentralen Speicher dar, allerdings wird und fast alles an Arbeit abgenommen.

```
User
    ident Text
    name Text
    password Text Maybe --password ist nicht zwingend gesetzt
    lastLogin UTCTime
    UniqueUser ident    --ident ist eindeutig
    deriving Typeable   --deriving geht auch

Post
    content Text
    user UserId
    time UTCTime
```

Maybe ist eine Optionale Erweiterung eines jeden Typen. Hier kann dann das Datenbank-Feld leer (`NULL`) sein, oder einen Inhalt haben.

Verwaltung der Datenbank

Die Datenbank stellt unseren zentralen Speicher dar, allerdings wird und fast alles an Arbeit abgenommen.

```
User
    ident Text
    name Text
    password Text Maybe --password ist nicht zwingend gesetzt
    lastLogin UTCTime
    UniqueUser ident    --ident ist eindeutig
    deriving Typeable   --deriving geht auch

Post
    content Text
    user UserId
    time UTCTime
```

Auch weitere Restriktionen kann man angeben. Hier sagen wir, dass es nicht 2 User mit derselben Identifikation geben kann.
Außerdem können wir Typklassen deriven.

Verwaltung der Datenbank

Die Datenbank stellt unseren zentralen Speicher dar, allerdings wird und fast alles an Arbeit abgenommen.

```
User
    ident Text
    name Text
    password Text Maybe --password ist nicht zwingend gesetzt
    lastLogin UTCTime
    UniqueUser ident    --ident ist eindeutig
    deriving Typeable   --deriving geht auch

Post
    content Text
    user UserId
    time UTCTime
```

Auch können wir unsere selbstdefinierten Typen in anderen Typen wiederverwenden. Hier ist jeder Post genau einem User zugeordnet.

Verwaltung der Datenbank

Was bringt uns das ganze nun?

Verwaltung der Datenbank

Was bringt uns das ganze nun?

Hieraus wird generiert:

Verwaltung der Datenbank

Was bringt uns das ganze nun?

Hieraus wird generiert:

- Alle angegeben Records (in den richtigen Typklassen)

Verwaltung der Datenbank

Was bringt uns das ganze nun?

Hieraus wird generiert:

- Alle angegeben Records (in den richtigen Typklassen)
- Für jeden Record wird eine ID erstellt

Verwaltung der Datenbank

Was bringt uns das ganze nun?

Hieraus wird generiert:

- Alle angegeben Records (in den richtigen Typklassen)
- Für jeden Record wird eine ID erstellt
- Für jedes Feld werden Datenbank-Typen erstellt

Verwaltung der Datenbank

Was bringt uns das ganze nun?

Hieraus wird generiert:

- Alle angegeben Records (in den richtigen Typklassen)
- Für jeden Record wird eine ID erstellt
- Für jedes Feld werden Datenbank-Typen erstellt

Wir erhalten somit aus

```
User
    ident Text
    name Text
    password Text Maybe
    lastLogin UTCTime
    UniqueUser ident
    deriving Typeable
```

Verwaltung der Datenbank

Was bringt uns das ganze nun?

Hieraus wird generiert:

- Alle angegeben Records (in den richtigen Typklassen)
- Für jeden Record wird eine ID erstellt
- Für jedes Feld werden Datenbank-Typen erstellt

Wir erhalten somit aus

```
User
    ident Text
    name Text
    password Text Maybe
    lastLogin UTCTime
    UniqueUser ident
    deriving Typeable
```

```
data User = User
          { userIdent     :: Text
          , userName      :: Text
          , userPassword  :: Maybe Text
          , userLastlogin :: UTCTime
          }

type UserId = Int

UserIdent     :: EntityField User Text
UserName      :: EntityField User Text
UserPassword  :: EntityField User (Maybe Text)
UserLastlogin :: EntityField User UTCTime
```

Verwaltung der Datenbank

Was bringt uns das ganze nun?

Hieraus wird generiert:

- Alle angegeben Records (in den richtigen Typklassen)
- Für jeden Record wird eine ID erstellt
- Für jedes Feld werden Datenbank-Typen erstellt

Wir erhalten somit aus

```
User
    ident Text
    name Text
    password Text Maybe
    lastLogin UTCTime
    UniqueUser ident
    deriving Typeable
```

```
data User = User
          { userIdent     :: Text
          , userName      :: Text
          , userPassword  :: Maybe Text
          , userLastlogin :: UTCTime
          }

type UserId = Int

UserIdent     :: EntityField User Text
UserName      :: EntityField User Text
UserPassword  :: EntityField User (Maybe Text)
UserLastlogin :: EntityField User UTCTime
```

Um den Rest kümmert sich Yesod. Wir müssen dies nachher nur noch benutzen.

Routen

Als nächstes überlegen wir uns, wie wir festlegen, was bei welchem Aufruf geschehen soll.

Routen

Als nächstes überlegen wir uns, wie wir festlegen, was bei welchem Aufruf geschehen soll.

Wir hatten eine URL der Form

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

und müssen uns nur noch um den markierten Part kümmern.

Routen

Als nächstes überlegen wir uns, wie wir festlegen, was bei welchem Aufruf geschehen soll.

Wir hatten eine URL der Form

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

und müssen uns nur noch um den markierten Part kümmern.

Also erstellen wir auch hier eine (Text-)Datei um Yesod zu sagen, was wo aufgerufen wird:

```
/static StaticR Static appStatic
/auth   AuthR   Auth   getAuth

/favicon.ico FaviconR GET
/robots.txt RobotsR GET

/ HomeR GET
/posts PostR GET
/edit/#Int64 PostEditR GET POST
```

Routen

Als nächstes überlegen wir uns, wie wir festlegen, was bei welchem Aufruf geschehen soll.

Wir hatten eine URL der Form

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

und müssen uns nur noch um den markierten Part kümmern.

Also erstellen wir auch hier eine (Text-)Datei um Yesod zu sagen, was wo aufgerufen wird:

```
/static StaticR Static appStatic
/auth   AuthR   Auth   getAuth

/favicon.ico FaviconR GET
/robots.txt RobotsR GET

/ HomeR GET
/posts PostR GET
/edit/#Int64 PostEditR GET POST
```

2 Spezielle Routen.
Eine für statische Inhalte (Bilder, Scripte, ..) und eine für die Authentifizierung von Usern.

Routen

Als nächstes überlegen wir uns, wie wir festlegen, was bei welchem Aufruf geschehen soll.

Wir hatten eine URL der Form

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

und müssen uns nur noch um den markierten Part kümmern.

Also erstellen wir auch hier eine (Text-)Datei um Yesod zu sagen, was wo aufgerufen wird:

```
/static StaticR Static appStatic
/auth   AuthR   Auth   getAuth

/favicon.ico FaviconR GET
/robots.txt RobotsR GET

/ HomeR GET
/posts PostR GET
/edit/#Int64 PostEditR GET POST
```

2 Routen für einzelne „Dateien“:
`favicon.ico` ist das Icon in Bookmarks
`robots.txt` enthält Anweisungen für Suchmaschienen

Routen

Als nächstes überlegen wir uns, wie wir festlegen, was bei welchem Aufruf geschehen soll.

Wir hatten eine URL der Form

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

und müssen uns nur noch um den markierten Part kümmern.

Also erstellen wir auch hier eine (Text-)Datei um Yesod zu sagen, was wo aufgerufen wird:

```
/static StaticR Static appStatic
/auth   AuthR   Auth   getAuth

/favicon.ico FaviconR GET
/robots.txt RobotsR GET

/ HomeR GET
/posts PostR GET
/edit/#Int64 PostEditR GET POST
```

Die eigentlichen Routen unserer Applikation.

Routen

Als nächstes überlegen wir uns, wie wir festlegen, was bei welchem Aufruf geschehen soll.

Wir hatten eine URL der Form

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

und müssen uns nur noch um den markierten Part kümmern.

Also erstellen wir auch hier eine (Text-)Datei um Yesod zu sagen, was wo aufgerufen wird:

```
/static StaticR Static appStatic
/auth   AuthR   Auth   getAuth

/favicon.ico FaviconR GET
/robots.txt RobotsR GET

/ HomeR GET
/posts PostR GET
/edit/#Int64 PostEditR GET POST
```

Wir bestimmen welche Route mit welchen Variablen...

Routen

Als nächstes überlegen wir uns, wie wir festlegen, was bei welchem Aufruf geschehen soll.

Wir hatten eine URL der Form

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

und müssen uns nur noch um den markierten Part kümmern.

Also erstellen wir auch hier eine (Text-)Datei um Yesod zu sagen, was wo aufgerufen wird:

```
/static StaticR Static appStatic
/auth   AuthR   Auth   getAuth

/favicon.ico FaviconR GET
/robots.txt RobotsR GET

/ HomeR GET
/posts PostR GET
/edit/#Int64 PostEditR GET POST
```

... durch welchen „Handler“...

Routen

Als nächstes überlegen wir uns, wie wir festlegen, was bei welchem Aufruf geschehen soll.

Wir hatten eine URL der Form

`protocol://subdomain.domain.tld/path/to/resource/item?param=value#anchor`

und müssen uns nur noch um den markierten Part kümmern.

Also erstellen wir auch hier eine (Text-)Datei um Yesod zu sagen, was wo aufgerufen wird:

```
/static StaticR Static appStatic
/auth   AuthR   Auth   getAuth

/favicon.ico FaviconR GET
/robots.txt RobotsR GET

/ HomeR GET
/posts PostR GET
/edit/#Int64 PostEditR GET POST
```

... mit welcher HTTP-Methode verstanden werden.

Routen

Die Funktionen die hierzu aufgerufen werden haben jeweils die Form:
`methodHandlerR`

Routen

Die Funktionen die hierzu aufgerufen werden haben jeweils die Form:
`methodHandlerR`
Somit müssen wir nur noch diese Funktion ausfüllen und genau einen Fall abfangen. So bauen wir unsere Applikation seite für Seite auf.

Routen

Die Funktionen die hierzu aufgerufen werden haben jeweils die Form: `methodHandlerR`
Somit müssen wir nur noch diese Funktion ausfüllen und genau einen Fall abfangen. So bauen wir unsere Applikation seite für Seite auf.

Mögliche Signaturen sieht dann so aus:

```
getHome       :: Handler Html
getPostEditR  :: Int64 -> Handler Html
postPostEditR :: Int64 -> Handler Html
```

Routen

Die Funktionen die hierzu aufgerufen werden haben jeweils die Form:
`methodHandlerR`
Somit müssen wir nur noch diese Funktion ausfüllen und genau einen Fall abfangen. So bauen wir unsere Applikation seite für Seite auf.

Mögliche Signaturen sieht dann so aus:

```
getHome       :: Handler Html
getPostEditR  :: Int64 -> Handler Html
postPostEditR :: Int64 -> Handler Html
```

`Handler` ist hierbei eine von Yesod bereitgestellte Monade mit der wir uns im Folgenden genauer beschäftigen.

Handler-Monade

Wir hatten letzte Woche bereits Monad-Transformer erklärt. Hierbei handelt es sich um genau so einen.

Handler-Monade

Wir hatten letzte Woche bereits Monad-Transformer erklärt. Hierbei handelt es sich um genau so einen.
In der Handler-Monade stecken zwei wesentliche Monaden, die wir benutzen wollen:

- `IO` in die wir uns mittels des gewohnten `liftIO` hineinheben
- Die Datenbank-Monade, in die wir mit `runDB` reinkommen

Handler-Monade

Wir hatten letzte Woche bereits Monad-Transformer erklärt. Hierbei handelt es sich um genau so einen.
In der Handler-Monade stecken zwei wesentliche Monaden, die wir benutzen wollen:

- `IO` in die wir uns mittels des gewohnten `liftIO` hineinheben
- Die Datenbank-Monade, in die wir mit `runDB` reinkommen

Somit sehen wir schon, dass - alleine durch die Typen - Aktionen am Model abgetrennt sind.

Handler-Monade

Wir hatten letzte Woche bereits Monad-Transformer erklärt. Hierbei handelt es sich um genau so einen.
In der Handler-Monade stecken zwei wesentliche Monaden, die wir benutzen wollen:

- `IO` in die wir uns mittels des gewohnten `liftIO` hineinheben
- Die Datenbank-Monade, in die wir mit `runDB` reinkommen

Somit sehen wir schon, dass - alleine durch die Typen - Aktionen am Model abgetrennt sind.

In dieser Monade schreiben wir auch später unsere gesamte Logik.

HTML-Generation

Da wir nun die Struktur unseres Models festgelegt haben, kommen wir nun zur Ausgabe.

HTML-Generation

Da wir nun die Struktur unseres Models festgelegt haben, kommen wir nun zur Ausgabe.

Das Problem ist, dass HTML nicht problemlos kombinierbar ist. Selbst, wenn wir einen Container haben, kann dieser auf eine Javascript-Bilbiothek angewiesen sein, die wir an dieser Stelle nicht definiren sollten.

HTML-Generation

Da wir nun die Struktur unseres Models festgelegt haben, kommen wir nun zur Ausgabe.

Das Problem ist, dass HTML nicht problemlos kombinierbar ist. Selbst, wenn wir einen Container haben, kann dieser auf eine Javascript-Bilbiothek angewiesen sein, die wir an dieser Stelle nicht definiren sollten.

Yesod löst dieses Problem über sogenannte „Widgets“.

HTML-Generation

Da wir nun die Struktur unseres Models festgelegt haben, kommen wir nun zur Ausgabe.

Das Problem ist, dass HTML nicht problemlos kombinierbar ist. Selbst, wenn wir einen Container haben, kann dieser auf eine Javascript-Bilbiothek angewiesen sein, die wir an dieser Stelle nicht definiren sollten.

Yesod löst dieses Problem über sogenannte „Widgets“.
Am Ende wird ein Layouter aufgerufen, der aus diesen (ineinandergesteckten) Widgets den HTML-Code generiert.

HTML-Generation

Da wir nun die Struktur unseres Models festgelegt haben, kommen wir nun zur Ausgabe.

Das Problem ist, dass HTML nicht problemlos kombinierbar ist. Selbst, wenn wir einen Container haben, kann dieser auf eine Javascript-Bilbiothek angewiesen sein, die wir an dieser Stelle nicht definiren sollten.

Yesod löst dieses Problem über sogenannte „Widgets“.
Am Ende wird ein Layouter aufgerufen, der aus diesen (ineinandergesteckten) Widgets den HTML-Code generiert.

Natürlich gibt es hier schon etwas vordefiniertes von Yesod:

```
defaultLayout :: PageContent (Route a) -> GHandler sub a Content
```

`defaultLayout` bekommt im Prinzip nur etwas „Widget-Ähnliches“ und generiert daraus eine Antwort.

HTML-Generation

Um Widgets zu erstellen gibt es auch Automatismen. Dieses wird über QuasiQuoter in Haskell implementiert (welche Haskell-Code generieren):

HTML-Generation

Um Widgets zu erstellen gibt es auch Automatismen. Dieses wird über QuasiQuoter in Haskell implementiert (welche Haskell-Code generieren):

```
let html = [hamlet|
  <h1>Hello World!
  <p>
    Willkommen
|]
```

wird zu

```
<h1>Hello World!</h1>
<p>Willkommen</p>
```

HTML-Generation

Um Widgets zu erstellen gibt es auch Automatismen. Dieses wird über QuasiQuoter in Haskell implementiert (welche Haskell-Code generieren):

```
let html = [hamlet|
  <h1>Hello World!
  <p>
    Willkommen
|]
```

wird zu

```
<h1>Hello World!</h1>
<p>Willkommen</p>
```

Der QuasiQuoter „Hamlet“ kümmert sich um das Schließen und verschachteln von Tags. Wir müssen diese einfach nur „richtig“ einrücken.

HTML-Generation

Um Widgets zu erstellen gibt es auch Automatismen. Dieses wird über QuasiQuoter in Haskell implementiert (welche Haskell-Code generieren):

```
let html = [hamlet|
  <h1>Hello World!
  <p>
    Willkommen
|]
```

wird zu

```
<h1>Hello World!</h1>
<p>Willkommen</p>
```

Der QuasiQuoter „Hamlet“ kümmert sich um das Schließen und verschachteln von Tags. Wir müssen diese einfach nur „richtig“ einrücken.
Alternative können wir diese auch manuell schließen.

HTML-Generation

Um Widgets zu erstellen gibt es auch Automatismen. Dieses wird über QuasiQuoter in Haskell implementiert (welche Haskell-Code generieren):

```
let html = [hamlet|
  <h1>Hello World!
  <p>
    Willkommen
|]
```

wird zu

```
<h1>Hello World!</h1>
<p>Willkommen</p>
```

Der QuasiQuoter „Hamlet“ kümmert sich um das Schließen und verschachteln von Tags. Wir müssen diese einfach nur „richtig“ einrücken.
Alternative können wir diese auch manuell schließen.

Variablen kann man so ebenso benutzen, wie automatisch korrekte Links zu generieren:

HTML-Generation

Um Widgets zu erstellen gibt es auch Automatismen. Dieses wird über QuasiQuoter in Haskell implementiert (welche Haskell-Code generieren):

```
let html = [hamlet|
  <h1>Hello World!
  <p>
    Willkommen
|]
```

wird zu

```
<h1>Hello World!</h1>
<p>Willkommen</p>
```

Der QuasiQuoter „Hamlet“ kümmert sich um das Schließen und verschachteln von Tags. Wir müssen diese einfach nur „richtig“ einrücken.
Alternative können wir diese auch manuell schließen.

Variablen kann man so ebenso benutzen, wie automatisch korrekte Links zu generieren:

```
let html name = [hamlet|
  <h1>Hello #{name}!
  <p><a href=@{AuthR LoginR}>Log in!</a>
|]
```

```
<h1>Hello Foobar!</h1>
<p><a href=/auth/login>Log in!</a></p>
```

wenn es mit `html "Foobar"` aufgerufen wird.

HTML-Generation

Um Widgets zu erstellen gibt es auch Automatismen. Dieses wird über QuasiQuoter in Haskell implementiert (welche Haskell-Code generieren):

```
let html = [hamlet|
  <h1>Hello World!
  <p>
    Willkommen
|]
```

wird zu

```
<h1>Hello World!</h1>
<p>Willkommen</p>
```

Der QuasiQuoter „Hamlet“ kümmert sich um das Schließen und verschachteln von Tags. Wir müssen diese einfach nur „richtig“ einrücken.
Alternative können wir diese auch manuell schließen.

Variablen kann man so ebenso benutzen, wie automatisch korrekte Links zu generieren:

```
let html name = [hamlet|
  <h1>Hello #{name}!
  <p><a href=@{AuthR LoginR}>Log in!</a>
|]
```

```
<h1>Hello Foobar!</h1>
<p><a href=/auth/login>Log in!</a></p>
```

wenn es mit `html "Foobar"` aufgerufen wird.

All dies ist automatisch „sicher“, da Hamlet gefährliche Zeichen in Variablen automatisch ersetzt.

Erster Handler

Eine simple Hello-World-Begrüßung wäre z.B.:

```
getHomeR :: Handler Html
getHomeR = defaultLayout $ toWidget [hamlet|
  <h1>Hello World!
      |]
```

Erster Handler

Eine simple Hello-World-Begrüßung wäre z.B.:

```
getHomeR :: Handler Html
getHomeR = defaultLayout $ toWidget [hamlet|
  <h1>Hello World!
      |]
```

Dies ist schon alles, was wir schreiben müssen, um eine Anfrage zu beantworten.

Handlergeneration

Damit es noch einfacher ist und wir fehler vermeiden, kann Yesod auch automatisch korrekte Handler generieren, wenn wir eine Route anlegen.

Handlergeneration

Damit es noch einfacher ist und wir fehler vermeiden, kann Yesod auch automatisch korrekte Handler generieren, wenn wir eine Route anlegen.

Hierzu machen wir lediglich ein `yesod add-handler` und beantworten die Fragen:

```
Name of route (without trailing R): New
Enter route pattern (ex: /entry/#EntryId): /newroute
Enter space-separated list of methods (ex: GET POST): GET
```

Handlergeneration

Damit es noch einfacher ist und wir fehler vermeiden, kann Yesod auch automatisch korrekte Handler generieren, wenn wir eine Route anlegen.

Hierzu machen wir lediglich ein `yesod add-handler` und beantworten die Fragen:

```
Name of route (without trailing R): New
Enter route pattern (ex: /entry/#EntryId): /newroute
Enter space-separated list of methods (ex: GET POST): GET
```

Dies macht die folgenden Dinge:

- legt eine Datei „Handler/New.hs“ an
- fügt einen import in die „Application.hs“ hinzu
- fügt das neue Modul in die cabal-File ein
- fügt die angegebene Route in die „models/route“-Datei hinzu

Handlergeneration

Der Inhalt der „Handler/New.hs“ ist danach:

```
module Handler.New where

import Import

getNewR :: Handler Html
getNewR = error "Not yet implemented: getNewR"
```

Handlergeneration

Der Inhalt der „Handler/New.hs“ ist danach:

```
module Handler.New where

import Import

getNewR :: Handler Html
getNewR = error "Not yet implemented: getNewR"
```

Hier können wir die Fehlermeldung einfach durch z.b. unser Hello-World-Widget austauschen.

Datenbankabfragen

Nun können wir Dinge schon darstellen und Anfragen beantworten. Wie bekommen wir nun aber Daten aus der Datenbank heraus?

Datenbankabfragen

Nun können wir Dinge schon darstellen und Anfragen beantworten. Wie bekommen wir nun aber Daten aus der Datenbank heraus?

Eben hatten wir schon gesagt, dass wir mittels `runDB` in die Datenbank-Monade kommen. Hier steht uns unter anderem zur Verfügung:

```
selectList :: (MonadIO m, PersistEntity val,
               PersistQuery backend,
               PersistEntityBackend val ~ backend)
           => [Filter val]
           -> [SelectOpt val]
           -> ReaderT backend m [Entity val]
```

Datenbankabfragen

Nun können wir Dinge schon darstellen und Anfragen beantworten. Wie bekommen wir nun aber Daten aus der Datenbank heraus?

Eben hatten wir schon gesagt, dass wir mittels `runDB` in die Datenbank-Monade kommen. Hier steht uns unter anderem zur Verfügung:

```
selectList :: (MonadIO m, PersistEntity val,
               PersistQuery backend,
               PersistEntityBackend val ~ backend)
           => [Filter val]
           -> [SelectOpt val]
           -> ReaderT backend m [Entity val]
```

Keine Panik!

Interessant für uns sind nur die ersten zwei Parameter. Wir können eine Liste von Filtern und eine Liste von Optionen angeben.

Interessant für uns sind nur die ersten zwei Parameter. Wir können eine Liste von Filtern und eine Liste von Optionen angeben.

Filter sind z.B.:

```
    [ PostContent <-. "needle"    --post-content contains "needle"
    , PostLikes >=. 100 ]         --and likes are over 100
||. [ PostUser ==. userid ]       --or user is the one with the given userid
```

Interessant für uns sind nur die ersten zwei Parameter. Wir können eine Liste von Filtern und eine Liste von Optionen angeben.

Filter sind z.B.:

```
    [ PostContent <-. "needle"    --post-content contains "needle"
    , PostLikes >=. 100 ]         --and likes are over 100
||. [ PostUser ==. userid ]       --or user is the one with the given userid
```

und Optionen:

```
-- only the first 50 posts
-- sorted by Likes
-- in descending order
[Desc PostLikes, LimitTo 50]
```

Wir erhalten damit eine Liste von Ergebnissen (vom Type `Entity`).

Wir erhalten damit eine Liste von Ergebnissen (vom Type `Entity`).
Eine `Entity` besteht aus

```
data Entity record = PersistEntity record =>
                     Entity { entityKey :: Key record
                            , entityVal :: record }
```

Wir erhalten damit eine Liste von Ergebnissen (vom Type `Entity`).
Eine `Entity` besteht aus

```
data Entity record = PersistEntity record =>
                     Entity { entityKey :: Key record
                            , entityVal :: record }
```

Wir bekommen also die interne Id, die wir z.B. für Updates brauchen und unsere Datenstruktur selbst. Konkret könnte das also so aussehen:

```
getHomeR :: Handler Html
getHomeR = do
  posts <- runDB $ selectList [] [LimitTo 50]
  defaultLayout $ [whamlet|
    <h1>last 50 posts
    <ul>
      $forall Entity pid (Post c u ts) <- posts
        <li>#{c}<br>
            Posted at #{show ts}
            <a href=@{PostEditR pid}>Edit!
|]
```

Wir erhalten damit eine Liste von Ergebnissen (vom Type `Entity`).
Eine `Entity` besteht aus

```
data Entity record = PersistEntity record =>
                     Entity { entityKey :: Key record
                            , entityVal :: record }
```

Wir bekommen also die interne Id, die wir z.B. für Updates brauchen und unsere Datenstruktur selbst. Konkret könnte das also so aussehen:

```
getHomeR :: Handler Html
getHomeR = do
  posts <- runDB $ selectList [] [LimitTo 50]
  defaultLayout $ [whamlet|
    <h1>last 50 posts
    <ul>
      $forall Entity pid (Post c u ts) <- posts
        <li>#{c}<br>
            Posted at #{show ts}
            <a href=@{PostEditR pid}>Edit!
|]
```

Alleine über Typinferenz wird diese Abfrage festgelegt und wir brauchen uns nicht mehr mit SQL herumschlagen.

Formulare

Jetzt wissen wir, wie man Daten anzeigt, aber noch nicht, wie man sie in die Datenbank bekommt.

Formulare

Jetzt wissen wir, wie man Daten anzeigt, aber noch nicht, wie man sie in die Datenbank bekommt.

Dies geschieht im HTML-Kontext zumeist über Formulare. Hierzu gibt es in Yesod 2 Wege:

- Applikativ (d.h. mit `Applicative`)

Formulare

Jetzt wissen wir, wie man Daten anzeigt, aber noch nicht, wie man sie in die Datenbank bekommt.

Dies geschieht im HTML-Kontext zumeist über Formulare. Hierzu gibt es in Yesod 2 Wege:

- Applikativ (d.h. mit `Applicative`)
  Hier wird der Code für uns automatisch generiert, aber wir haben kaum Einfluss auf die Gestaltung.

Formulare

Jetzt wissen wir, wie man Daten anzeigt, aber noch nicht, wie man sie in die Datenbank bekommt.

Dies geschieht im HTML-Kontext zumeist über Formulare. Hierzu gibt es in Yesod 2 Wege:

- Applikativ (d.h. mit `Applicative`)
  Hier wird der Code für uns automatisch generiert, aber wir haben kaum Einfluss auf die Gestaltung.
- Monadisch (d.h. mit `Monad`)

Formulare

Jetzt wissen wir, wie man Daten anzeigt, aber noch nicht, wie man sie in die Datenbank bekommt.

Dies geschieht im HTML-Kontext zumeist über Formulare. Hierzu gibt es in Yesod 2 Wege:

- Applikativ (d.h. mit `Applicative`)
  Hier wird der Code für uns automatisch generiert, aber wir haben kaum Einfluss auf die Gestaltung.
- Monadisch (d.h. mit `Monad`)
  Hier können wir einzelne Felder selektieren, müssen aber seperat das HTML generieren, welches im Browser angezeigt wird

Formulare

Jetzt wissen wir, wie man Daten anzeigt, aber noch nicht, wie man sie in die Datenbank bekommt.

Dies geschieht im HTML-Kontext zumeist über Formulare. Hierzu gibt es in Yesod 2 Wege:

- Applikativ (d.h. mit `Applicative`)
  Hier wird der Code für uns automatisch generiert, aber wir haben kaum Einfluss auf die Gestaltung.
- Monadisch (d.h. mit `Monad`)
  Hier können wir einzelne Felder selektieren, müssen aber seperat das HTML generieren, welches im Browser angezeigt wird

Wir werden vorerst nur die applikative Syntax benutzen. Die monadische Variante ist im Yesod-Buch aber sehr gut erklärt und ggf. einfach zu adaptieren.

Formulare

Kommen wir zunächst zu der Syntax für die applikative Schreibweise.

Formulare

Kommen wir zunächst zu der Syntax für die applikative Schreibweise.
Ein Beispielformular für unseren Post könnte in etwa so Aussehen:

```
postForm :: UserId -> Form Post
postForm u = renderDivs $ Post
    <$> areq textField "Post" (Just "Post content")
    <*> pure u
    <*> lift (liftIO getCurrentTime)
```

Formulare

Kommen wir zunächst zu der Syntax für die applikative Schreibweise.
Ein Beispielformular für unseren Post könnte in etwa so Aussehen:

```
postForm :: UserId -> Form Post
postForm u = renderDivs $ Post
    <$> areq textField "Post" (Just "Post content")
    <*> pure u
    <*> lift (liftIO getCurrentTime)
```

Hier generieren wir einen Rant aus

- einer benötigten einzeiligem TextFeld mit Default-Wert „Post content“
- dem der Funktion übergebenen UserId
- der aktuellen Uhrzeit (die nicht vom Client, sondern vom Server kommt)

Formulare

In der applikativen Notation haben wir vier Möglichkeiten Werte zu bekommen:

Formulare

In der applikativen Notation haben wir vier Möglichkeiten Werte zu bekommen:

- Benötigte Werte vom Client (`areq`)

Formulare

In der applikativen Notation haben wir vier Möglichkeiten Werte zu bekommen:

- Benötigte Werte vom Client (`areq`)
- Optionale Werte vom Client (`aopt`)

Formulare

In der applikativen Notation haben wir vier Möglichkeiten Werte zu bekommen:

- Benötigte Werte vom Client (`areq`)
- Optionale Werte vom Client (`aopt`)
- Konstanten (`pure constant`)

Formulare

In der applikativen Notation haben wir vier Möglichkeiten Werte zu bekommen:

- Benötigte Werte vom Client (`areq`)
- Optionale Werte vom Client (`aopt`)
- Konstanten (`pure constant`)
- Daten aus der Handler-Monade (`lift`)

Formulare

In der applikativen Notation haben wir vier Möglichkeiten Werte zu bekommen:

- Benötigte Werte vom Client (`areq`)
- Optionale Werte vom Client (`aopt`)
- Konstanten (`pure constant`)
- Daten aus der Handler-Monade (`lift`)

Natürlich können wir in der Handler-Monade dann durch `liftIO` beliebige Funktionen ausführen.

Formulare

Nun haben wir ein Formular definiert. Dieses wird sowohl für das Anzeigen beim Client verwendet, als auch für die Validierung auf dem Server.

Formulare

Nun haben wir ein Formular definiert. Dieses wird sowohl für das Anzeigen beim Client verwendet, als auch für die Validierung auf dem Server.
Hierzu gibt es die Funktion

```
(widget, enctype) <- generateFormPost postForm
```

um ein Formular zu generieren.

Formulare

Nun haben wir ein Formular definiert. Dieses wird sowohl für das Anzeigen beim Client verwendet, als auch für die Validierung auf dem Server.
Hierzu gibt es die Funktion

```
(widget, enctype) <- generateFormPost postForm
```

um ein Formular zu generieren.

`widget` enthält den HTML-Teil, den wir mittels `defaultLayout` rendern können. `enctype` enthält den Encoding-Type, der im äußeren Formular angegeben werden muss.

Eine fertige Seite würde somit wie folgt aussehen:

```
getPostNewR :: Handler Html
getPostNewR = do
  (postWidget, postEnctype) <- generateFormPost postForm
  defaultLayout $ do
    [whamlet|
      <h1>Post
      <form method=post action=@{PostNewR} enctype=#{postEnctype}>
        ^{postWidget}
        <button>Post!
    |]
```

Eine fertige Seite würde somit wie folgt aussehen:

```
getPostNewR :: Handler Html
getPostNewR = do
  (postWidget, postEnctype) <- generateFormPost postForm
  defaultLayout $ do
    [whamlet|
      <h1>Post
      <form method=post action=@{PostNewR} enctype=#{postEnctype}>
        ^{postWidget}
        <button>Post!
    |]
```

Wir sehen hier, dass im Hamlet mittels `^{postWidget}` das Widget direkt eingebunden werden kann. Wir müssen nur noch das äußere `<form>`-Konstrukt definieren und einen Button zum Absenden hinzufügen.

Eine fertige Seite würde somit wie folgt aussehen:

```
getPostNewR :: Handler Html
getPostNewR = do
  (postWidget, postEnctype) <- generateFormPost postForm
  defaultLayout $ do
    [whamlet|
      <h1>Post
      <form method=post action=@{PostNewR} enctype=#{postEnctype}>
        ^{postWidget}
        <button>Post!
    |]
```

Wir sehen hier, dass im Hamlet mittels `^{postWidget}` das Widget direkt eingebunden werden kann. Wir müssen nur noch das äußere `<form>`-Konstrukt definieren und einen Button zum Absenden hinzufügen.
Nach dem Abschicken wird die Route `PostNewR` mittels `post` aufgerufen, wo wir dann das Ergebnis bearbeiten müssen.

Formulare

Wenn wir ein Formular auswerten wollen, dann benutzen wir

```
((result,postWidget), postEnctype) <- runFormPost postForm
```

Formulare

Wenn wir ein Formular auswerten wollen, dann benutzen wir

```
((result,postWidget), postEnctype) <- runFormPost postForm
```

`result` ist hierbei das Ergebnis des Formulars. Falls irgendetwas nicht stimmt, dann bekommen wir gleich auch noch das (teilausgefüllte) Widget und den Enctype zurück um dem User das Formular erneut anzuzeigen.

Formulare

Wenn wir ein Formular auswerten wollen, dann benutzen wir

```
((result,postWidget), postEnctype) <- runFormPost postForm
```

`result` ist hierbei das Ergebnis des Formulars. Falls irgendetwas nicht stimmt, dann bekommen wir gleich auch noch das (teilausgefüllte) Widget und den Enctype zurück um dem User das Formular erneut anzuzeigen.
Außerdem ist `result` vom Typen `FormResult a`:

```
data FormResult a = FormMissing
                  | FormFailure [Text]
                  | FormSuccess a
```

Formulare

Wenn wir ein Formular auswerten wollen, dann benutzen wir

```
((result,postWidget), postEnctype) <- runFormPost postForm
```

`result` ist hierbei das Ergebnis des Formulars. Falls irgendetwas nicht stimmt, dann bekommen wir gleich auch noch das (teilausgefüllte) Widget und den Enctype zurück um dem User das Formular erneut anzuzeigen.
Außerdem ist `result` vom Typen `FormResult a`:

```
data FormResult a = FormMissing
                  | FormFailure [Text]
                  | FormSuccess a
```

Für die drei möglichen Fälle:

- Keine Formulardaten vorhanden
- Fehlermeldungen
- Erfolg

Formulare

Normalerweise macht man ein case über das result:

```
postPostNewR :: Handler Html
postPostNewR = do
  ((result,postWidget), postEnctype) <- runFormPost postForm
  let again err = defaultLayout $ do
       [whamlet|
      <h1>Post
      <h2>Fehler:
      <p>#{err}
      <form method=post action=@{PostNewR} enctype=#{postEnctype}>
        ^{postWidget}
        <button>Post!
       |]
  case result of
    FormSuccess post    -> do --put into database
                              _ <- runDB $ insert post
                              redirect getHomeR --and redirect home
    FormFailure (err:_) -> again err
    _ -> again "Invalid input"
```

Formulare

Normalerweise macht man ein case über das `result`:

```
postPostNewR :: Handler Html
postPostNewR = do
  ((result,postWidget), postEnctype) <- runFormPost postForm
  let again err = defaultLayout $ do
       [whamlet|
      <h1>Post
      <h2>Fehler:
      <p>#{err}
      <form method=post action=@{PostNewR} enctype=#{postEnctype}>
        ^{postWidget}
        <button>Post!
       |]
  case result of
    FormSuccess post    -> do --put into database
                              _ <- runDB $ insert post
                              redirect getHomeR --and redirect home
    FormFailure (err:_) -> again err
    _ -> again "Invalid input"
```

Auch hier sehen wir, dass über die Typen für die Datenbank schon klar ist, was sie tun soll und das einfügen fällt prinzipiell auf ein `insert object` zusammen.

Zugriffsbeschränkungen

Für manche Bereiche einer Seite wollen wir allerdings Zugangsbeschränkungen. Dies wird über Regeln in der „Foundation.hs“ gemacht.

Zugriffsbeschränkungen

Für manche Bereiche einer Seite wollen wir allerdings Zugangsbeschränkungen. Dies wird über Regeln in der „Foundation.hs“ gemacht.

Autogeneriert wird hierzu

```
-- Routes not requiring authentication.
isAuthorized (AuthR _) _ = return Authorized
-- Default to Authorized for now.
isAuthorized _ _ = return Authorized
```

Zugriffsbeschränkungen

Für manche Bereiche einer Seite wollen wir allerdings Zugangsbeschränkungen. Dies wird über Regeln in der „Foundation.hs“ gemacht.

Autogeneriert wird hierzu

```
-- Routes not requiring authentication.
isAuthorized (AuthR _) _ = return Authorized
-- Default to Authorized for now.
isAuthorized _ _ = return Authorized
```

Der erste Parameter ist hier eine Route, der zweite Paremeter ein Bool, der für den Schreibzugriff steht.

Zugriffsbeschränkungen

Für manche Bereiche einer Seite wollen wir allerdings Zugangsbeschränkungen. Dies wird über Regeln in der „Foundation.hs“ gemacht.

Autogeneriert wird hierzu

```
-- Routes not requiring authentication.
isAuthorized (AuthR _) _ = return Authorized
-- Default to Authorized for now.
isAuthorized _ _ = return Authorized
```

Der erste Parameter ist hier eine Route, der zweite Paremeter ein Bool, der für den Schreibzugriff steht.
Wir wollen, dass jeder eingeloggte User posten und Post editieren kann. Also fügen wir hinzu:

```
isAuthorized (PostEditR _) _ = isUser
isAuthorized PostR _ = isUser
isUser = do
  mu <- maybeAuthId
  return $ case mu of
      Nothing -> AuthenticationRequired
      Just _  -> Authorized
```

Zugriffsbeschränkungen

Allerdings ist dies nur eine grobe Prüfung auf Dinge, die wir ohne die Datenbank wissen. Eine genaue Prüfung - etwa, dass ein User nur seine eigenen Posts editieren darf - muss im Handler gemacht werden.

Fazit

Wir haben einmal einen groben Überblick über die Funktionsweise von Webapplikationen im Allgemeinen und deren Umsetzung mit Yesod im speziellen gewonnen.

Fazit

Wir haben einmal einen groben Überblick über die Funktionsweise von Webapplikationen im Allgemeinen und deren Umsetzung mit Yesod im speziellen gewonnen.

Insbesondere betrachteten wir:

Fazit

Wir haben einmal einen groben Überblick über die Funktionsweise von Webapplikationen im Allgemeinen und deren Umsetzung mit Yesod im speziellen gewonnen.

Insbesondere betrachteten wir:

- URLs und Routen

Fazit

Wir haben einmal einen groben Überblick über die Funktionsweise von Webapplikationen im Allgemeinen und deren Umsetzung mit Yesod im speziellen gewonnen.

Insbesondere betrachteten wir:

- URLs und Routen
- Definition eines Datenbankmodells

Fazit

Wir haben einmal einen groben Überblick über die Funktionsweise von Webapplikationen im Allgemeinen und deren Umsetzung mit Yesod im speziellen gewonnen.

Insbesondere betrachteten wir:

- URLs und Routen
- Definition eines Datenbankmodells
- Abfragen von Informationen aus der Datenbank

Fazit

Wir haben einmal einen groben Überblick über die Funktionsweise von Webapplikationen im Allgemeinen und deren Umsetzung mit Yesod im speziellen gewonnen.

Insbesondere betrachteten wir:

- URLs und Routen
- Definition eines Datenbankmodells
- Abfragen von Informationen aus der Datenbank
- Generierung von HTML

Fazit

Wir haben einmal einen groben Überblick über die Funktionsweise von Webapplikationen im Allgemeinen und deren Umsetzung mit Yesod im speziellen gewonnen.

Insbesondere betrachteten wir:

- URLs und Routen
- Definition eines Datenbankmodells
- Abfragen von Informationen aus der Datenbank
- Generierung von HTML
- Generierung und Auswertung von Formularen

Fazit

Wir haben einmal einen groben Überblick über die Funktionsweise von Webapplikationen im Allgemeinen und deren Umsetzung mit Yesod im speziellen gewonnen.

Insbesondere betrachteten wir:

- URLs und Routen
- Definition eines Datenbankmodells
- Abfragen von Informationen aus der Datenbank
- Generierung von HTML
- Generierung und Auswertung von Formularen
- Authentifikation

Fazit

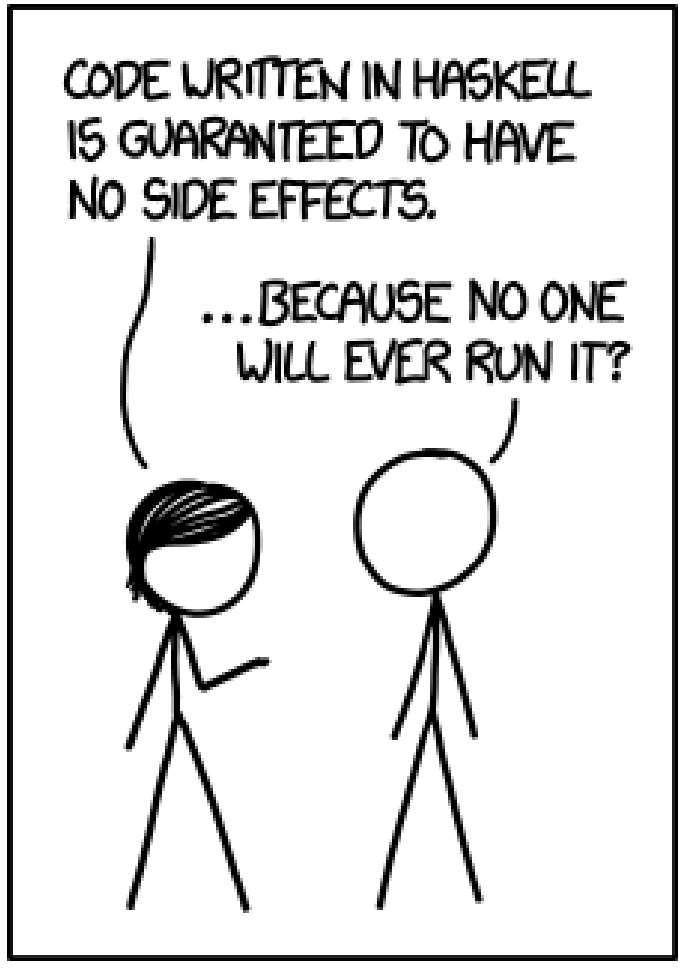
Wir haben einmal einen groben Überblick über die Funktionsweise von Webapplikationen im Allgemeinen und deren Umsetzung mit Yesod im speziellen gewonnen.

Insbesondere betrachteten wir:

- URLs und Routen
- Definition eines Datenbankmodells
- Abfragen von Informationen aus der Datenbank
- Generierung von HTML
- Generierung und Auswertung von Formularen
- Authentifikation

Fragen?

Demo



Hovertext: „The problem with Haskell is that it's a language built on lazy evaluation and nobody's actually called for it.“

https://xkcd.com/1312/